Microsoft®
BASIC Interpreter

for Premium SoftCard® IIe System for Apple® IIe

Reference Manual

Microsoft Corporation

Comments about this documentation may be sent to:

Microsoft Corporation
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

Document Number 8101A-527-00

Contents

4 Microsoft BASIC Functions 131

5 High-Resolution Graphics 163

Chapter 1
Introduction

Chapter 1
Introduction

Microsoft® BASIC Interpreter Release 5.27 is the most extensive implementation of BASIC available for microprocessors. It meets the requirements for the ANSI subset standard for BASIC, and supports many features rarely found in other BASIC interpreters. In addition, Microsoft BASIC Interpreter has sophisticated string-handling and structured programming features that are especially suited for application development. And Microsoft BASIC Interpreter is compatible with Microsoft BASIC Compiler. Microsoft BASIC Interpreter gives users what they want from BASIC — ease of use, plus the features that make a microcomputer perform like a minicomputer or large mainframe.

In 1975, Microsoft wrote the first BASIC interpreter for microcomputers. Today, Microsoft BASIC Interpreter has over 750,000 installations in over 20 operating environments. It's the BASIC you will find on all of the most popular microcomputers. Many users, manufacturers, and software vendors have written application programs in Microsoft BASIC. Standard Microsoft BASIC Interpreter features include:

16-digit precision: Three variable types (fast two-byte true integer variables, single precision variables, and double precision variables) provide 16-digit precision.

EDIT commands: Extensive editing commands let you edit individual program lines easily and efficiently, without reentering the entire line.

Built-in disk I/O statements: Provide fast and powerful disk I/O access.

PRINT USING: Greatly enhances programming convenience by making it easy to format output. Includes asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion.

WHILE/WEND: Gives BASIC a more structured organization. By putting a WHILE statement in front of and a WEND statement at the end of a loop, you can continuously execute the loop as long as a given condition is true.

AUTO and RENUM: AUTO generates a line number automatically after every carriage return. RENUM automatically renumbers lines in user-specified increments.

CHAIN and COMMON: Call in another BASIC program from disk and pass variables to it.

IF...THEN[...ELSE]: Extends the IF statement to provide for the negative case of IF.

Added operators: Microsoft BASIC provides AND, OR, XOR, EQV, IMP, and NOT.

Expanded user-defined functions: Can have multiple parameters.

1.1 How to Use this Manual

This manual has been specially prepared for use with Microsoft BASIC Interpreter Release 5.27, which is included in your Microsoft Premium SoftCard™ IIe package. It serves as both a user's guide and a technical reference, documenting both general information and detailed descriptions of Microsoft BASIC commands, statements, and functions.

This manual is not intended as a tutorial on BASIC. It is assumed that you have a working knowledge of the BASIC language. If you need more information on BASIC programming, refer to Section 1.3, "Resources for Learning BASIC."

This manual contains the following information:

Chapter 1 *Introduction*

Provides a brief description of the contents of this manual and the notation used in describing BASIC language syntax. Also includes a list of references for learning BASIC programming.

Chapter 2 *General Information about Microsoft BASIC Interpreter*

Explains how to get Microsoft BASIC up and running. Also briefly describes modes of operation, program format, special characters, data representation, and input editing.

Chapter 3 *Microsoft BASIC Commands and Statements*

Describes Microsoft BASIC commands and statements.

Chapter 4 *Microsoft BASIC Functions*

Describes Microsoft BASIC functions.

Chapter 5 *High-Resolution Graphics:*

Describes Microsoft GBASIC high-resolution graphic statements and commands.

Appendix A *Microsoft BASIC Interpreter and Applesoft: A Comparison*

Compares Microsoft BASIC Interpreter with Applesoft® BASIC.

Appendix B *Differences Between Microsoft BASIC Interpreter 5.27 and Earlier Releases*

Describes the features that are new to Microsoft BASIC in Release 5.27 and how BASIC Release 5.27 differs from previous releases.

Appendix C *Converting Programs to Microsoft BASIC*

Shows how to convert a program written in another BASIC to Microsoft BASIC.

Appendix D *Microsoft BASIC Disk I/O*

Explains disk I/O procedures for the user who is unfamiliar with disk I/O conventions and routines.

Appendix E *Microsoft BASIC Assembly Language Subroutines*

Discusses how to interface to assembly language subroutines with the USR function and the CALL statement.

Appendix F *Mathematical Functions*

Provides a set of formulas for math functions that are not built into Microsoft BASIC Interpreter.

Appendix G *Microsoft BASIC Floating-Point Representation*

Discusses how Microsoft BASIC handles floating-point operations.

Appendix H *ASCII Character Codes*

Lists the values of the ASCII character set.

Appendix I *Microsoft BASIC Reserved Words*

Lists Microsoft BASIC reserved words.

Appendix J *Error Codes and Error Messages*

Presents a detailed description of all the error messages in Microsoft BASIC and their possible causes.

1.2 Syntax Notation

Every language must be learned before it can be used. In this manual, special notation has been developed to show the differences in what you enter on the keyboard and what you see on the display screen. Examples of use are given at the end of this section.

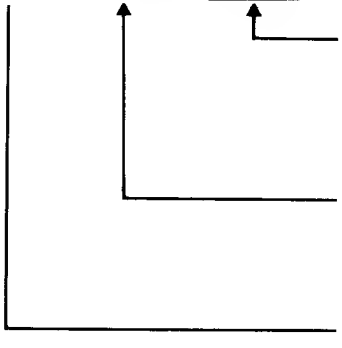
< >	Angle brackets indicate information that you enter. Angle brackets that enclose lowercase text are for entries that you must supply, such as a <filename>.
[ ]	Square brackets indicate that the enclosed entry is optional. The same rules about uppercase and lowercase text also apply.
{ }	Braces indicate a choice between two or more entries. At least one of the entries enclosed in braces must be chosen, unless the entries are also enclosed in square brackets.
|	Vertical bars separate choices within braces.
. . .	Ellipses indicate that an entry can be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

Examples

Command Line *Explanation*

SAVE <filespec> [{A|P}]



These two entries are optional as indicated by the square brackets. They also must be typed in as shown. The braces indicate that you must enter either the /A or the /P entry.

The lowercase filespec means you must supply the file specification (disk drive, filename and extension).

Capital letters indicate that the word must be entered exactly as shown.

1.3 Resources for Learning BASIC

This manual provides complete instructions for using Microsoft BASIC Interpreter. However, no teaching material for BASIC programming has been provided. If you are new to BASIC or need help in learning programming, we suggest you read one of the following:

Albrecht, Robert L., Finkel, LeRoy, and Brown, Jerry. *BASIC.* New York: Wiley Interscience, 2nd ed., 1978.

Billings, Karen and Moursund, David. *Are You Computer Literate?* Beaverton, Oregon: Dilithium Press, 1979.

Coan, James. *Basic BASIC.* Rochelle Park, N.J.: Hayden Book Company, 1978.

Dwyer, Thomas A. and Critchfield, Margot. *BASIC and the Personal Computer.* Reading, Mass.: Addison-Wesley Publishing Co., 1978.

Simon, David E. *BASIC From the Ground Up.* Rochelle Park, N.J.: Hayden Book Company, 1978.

Chapter 2

General Information about Microsoft BASIC

Chapter 2

General Information about Microsoft BASIC

GBASIC (file GBASIC.COM) is the CP/M® version of Microsoft BASIC Interpreter which includes all standard Applesoft extensions including high-resolution graphics. This file can be found on the SoftCard Master disk.

2.1 Initialization

To load and run GBASIC, bring up CP/M and wait for the A> prompt. Once the prompt appears, type the following:

```
GBASIC <RETURN>
```

The system will reply:

```
BASIC-80 REV 5.27
(SOFTCARD //e CP/M VERSION)
COPYRIGHT 1983 (C) BY MICROSOFT
CREATED: DD-MM-YY
xxxxx BYTES FREE
```

This sets the number of files that can be open at one time during execution of a BASIC program to three. It also sets the maximum record size at 128 bytes and allows the use of all RAM memory up to the start of FDOS (an arbitrary area in memory set by CP/M).

If you wish to change the memory configuration, the following command line format can be used for initialization in place of the simple GBASIC command.

```
GBASIC [<filespec>][/F:<number of files>]
[/M:<highest memory location>][/S:<maximum record size>]
```

The <filespec> option allows you to run a program after initialization is complete. <filespec> consists of a filename and optional filename extension. A default extension of .BAS is used if none is supplied and the filename is less than nine characters long. This allows BASIC programs to be executed in batch mode using the CP/M SUBMIT transient program. Such programs should include a SYSTEM statement (see Section 3.70) to induce return to CP/M command level when they have finished and allow the next program in the batch stream to execute.

The /F:<number of files> option sets the number of disk files that can be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes plus 128 bytes (or the number specified in the /S: option) of memory. If the /F option is omitted, the number of files defaults to 3. The <number of files> can be entered in decimal form (default condition), octal form (preceded by an &O), or hexadecimal form (preceded by an &H).

The /M:<highest memory location> option sets the highest memory location that will be used by GBASIC. In some cases, it is desirable to set the amount of memory well below the FDOS area to reserve space for assembly language subroutines. In all cases, <highest memory location> should be set below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used.

The /S:<maximum record size> option sets the maximum record size for use with random access files. Any whole number can be specified, including numbers larger than 128 (the default record size).

Here are a few examples of initialization options:

A>GBASIC PAYROLL.BAS	Use all memory and three files, load and execute PAYROLL.BAS.
A>GBASIC INVENT/F:6	Use all memory and six files, load and execute INVENT.BAS.
A>GBASIC /M:32768	Use the first 32K bytes of memory and three files.
A>GBASIC DATACK/F:2/M: &H9000	Use the first 36K bytes of memory, two files, and execute DATACK.BAS.

To return to CP/M, use the SYSTEM command. SYSTEM closes all files and then performs a CP/M warm start (reboots CP/M). See Section 3.70, "SYSTEM."

2.2 Operational Modes

When Microsoft BASIC is initialized, the prompt "Ok" is displayed. "Ok" means BASIC is at command level; that is, it is ready to accept commands. At this point, Microsoft BASIC can be used in either of two modes: direct mode or indirect mode.

In direct mode, BASIC statements and commands are not preceded by line numbers; they are executed as they are entered. Results of arithmetic and logical operations are displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

Indirect mode is used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

2.3 Screen Display Modes

There are four screen display modes in Microsoft BASIC:

Text display mode

Low-resolution graphics display mode

Mixed text display mode

High-resolution graphics display mode

The text display mode is the default screen display mode whenever you initialize BASIC. In this mode, BASIC displays only text characters on the screen.

Low-resolution graphics display mode allows graphics plotting on a screen grid measuring 40 rows by 48 columns (40x48). No text (ASCII) characters can be used. BASIC enters this mode through the GR command. The TEXT command returns BASIC to text display mode.

Mixed text display mode allows display of both graphics and text. Low-resolution graphics can be plotted on a 40x40 screen grid starting at the top of the screen. The bottom four screen lines are for reserved text entry. BASIC enters mixed text display mode by setting GR command to zero. The TEXT command returns BASIC to text display mode.

High-resolution graphics (GBASIC) mode allows graphics plotting on a 280x192 screen grid. No text (ASCII) characters can be used. High-resolution graphics mode is initiated by setting the HGR command to either 1 or 3. The TEXT command returns BASIC to text display mode.

2.4 CP/M File Naming Conventions

CP/M disk files are described by their file specifications or *filespecs*, for short. Filespecs are string expressions with the format:

[drive identifier:] <filename>[.filename extension]

The drive identifier option tells CP/M where to look for the file. <filename> tells CP/M which file to look for. The filename extension option is a label that tells CP/M what type of file it is. <filename> is the only required argument. Each of these arguments is described in the following sections.

2.4.1 Filename

CP/M filenames can be from one to eight characters in length, and can consist of either uppercase or lowercase alphanumeric characters, or a combination of both. However, if you use lowercase letters in a filename, the CP/M built-in command ERA will not recognize the filename. CP/M also does not recognize filenames longer than eight characters.

Examples of valid filenames:

PAYROLL ACNT4 A2400 Barb

Certain special characters cannot be used as filenames. These characters are:

= ? * < > . , ; : [ ]

In addition, no CONTROL characters can be used as filenames.

2.4.2 Filename Extension

A filename extension identifies the type of a file. For example, .ASM identifies an assembly language source file, whereas .BAS identifies a BASIC program source file. Filename extensions in CP/M consist of one to three characters and are preceded by a period. The filename can be made up of letters or numeric characters, or a combination of both. Most often, you will use one of the extensions listed in Table 2.1.

Table 2.1. Filename Extensions

Extension	Type
.ASM	Assembly language source file
.BAK	Backup file
.BAS	BASIC source file
.COM	Command file
.COB	COBOL source file
.DAT	Data file
.DOC	Text document file
.EXE	Executable file
.FOR	FORTRAN source file
.HEX	Intel HEX format object code file
.LIB	Library file
.MAC	Macro assembler source file (usually a subroutine used in assembly language programs)
.OBJ	Machine code (object file)
.PAS	Pascal source file
.PRN	Assembly language list file (PRINT file)
.REL	Relocatable machine-code program file
.TXT	Text file

Although other extensions can be used, this table lists most of the extensions you will use with CP/M.

The most common extension you will use is .BAS. .BAS is the default extension for LOAD, SAVE, MERGE, and RUN commands (if no other extension is given and the filename is less than eight characters long).

Examples of filename extensions:

```
APPLE3.TXT      ACCReciv.BAS      PROGRAM.4      POLS.C12
```

2.4.3 Disk Drive Identifier

Disk drives are identified in CP/M by capital letters. The first drive is the *primary drive* and is always identified by the letter A. Successive drives are identified in alphabetical order.

Disk drive identifiers precede the filename, and consist of the identifying letter (A-F) and a colon (:). The colon separates the disk drive identifier from the filename.

If no identifier is specified, CP/M searches the default drive (unless otherwise specified, the default drive is always drive A:). For example,

```
PROGRAM.BAS.
```

is assumed to be on a disk in drive A:.

2.5 Line Format

Microsoft BASIC program lines have the following format:

```
nnnnn BASIC statement [: BASIC statement...] [comment]
```

Microsoft BASIC program lines always begin with a line number (nnnnn) and end with a carriage return. A program line can contain a maximum of 255 characters. More than one BASIC statement can be placed on a line, but each must be separated from the last by a colon.

Program lines are ended by pressing the <RETURN> key. It is possible to extend a logical line over more than one physical line by entering CONTROL-J near the end of a physical line. CONTROL-J lets you continue typing a logical line on the next physical line without entering <RETURN>. The line is not terminated until you press <RETURN>.

Line numbers indicate the order in which the program lines are stored in memory. They are also used as references in branching and editing. Line numbers must be in the range 0 to 65529. A period (.) can be used in EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

2.6 Character Set

The Microsoft BASIC character set is composed of alphabetic, numeric, and special characters. These are the only characters that Microsoft BASIC recognizes. There are many other characters which can be displayed or printed, but have no particular meaning to Microsoft BASIC.

The Microsoft BASIC alphabetic characters include all the uppercase and lowercase letters of the alphabet. Numeric characters are the digits 0 through 9.

Table 2.2 lists the special characters and terminal keys that are recognized by Microsoft BASIC.

Table 2.2. Microsoft BASIC Character Set

Character	Name or Function
	Blank
=	Equal sign or assignment symbol
+	Plus sign
−	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
<ESCAPE>	Escapes edit mode subcommands (See Chapter 3)
<TAB>	Moves print position to next tab stop Tab stops are set at every eight columns
<RETURN>	Terminates input of a line

Table 2.3 lists the CONTROL characters that are used in Microsoft BASIC.

Table 2.3. Microsoft BASIC CONTROL Characters

CONTROL Character	Function
CONTROL-A	Enters edit mode on the line being typed.
CONTROL-B	Backslash.
CONTROL-C	Interrupts program execution and returns to BASIC command level.
CONTROL-G	Rings the bell (a beep from the speaker) at the console.
CONTROL-H	Backspace. Deletes the last character typed. Same as the ← key.
CONTROL-I	Tab. Tab stops are set at every eight columns. Same as the → key.
CONTROL-J	Linefeed. Moves cursor to the next physical line.
CONTROL-K	Right square bracket.
CONTROL-O	Halts program output while execution continues. A second CONTROL-O restarts output.
CONTROL-Q	Resumes program execution after a CONTROL-S has been executed.
CONTROL-R	Redisplays the line that is currently being typed.
CONTROL-S	Suspends program execution.
CONTROL-U	Deletes the line that is currently being typed.
CONTROL-X	Same as CONTROL-U.
CONTROL-Y	Permits recovery after <RESET> has been pressed.

2.7 Reserved Words

Reserved words are words that have special meaning in Microsoft BASIC. They include all BASIC commands, statements, function names, and operator names.

Always separate reserved words from data or other elements of a BASIC statement with spaces or other special characters, as allowed by the syntax. Reserved words cannot be used as variable names.

A complete list of Microsoft BASIC reserved words is given in Appendix I.

2.8 Constants

Constants are the actual values BASIC uses during program execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. There are five types of numeric constants:

Integer constants	Whole numbers between −32768 and +32767. Integer constants do not contain decimal points.
Fixed-point constants	Positive or negative real numbers; i.e., numbers that contain decimal points.

Floating-point constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constants is 10^{-38} to 10^{+38}.

Examples:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
```

(Double precision floating-point constants are denoted by the letter D instead of E.)

Hex constants

Hexadecimal numbers with the prefix &H.

Examples:

```
&H76
&H32F
```

Octal constants

Octal numbers with the prefix &O or &.

Examples:

```
&O347
&1234
```

Numeric constants can be either single precision or double precision numbers. Single precision numeric constants are stored with 6 digits of precision (plus the exponent) and printed with up to 6 digits of precision. Double precision numbers are stored with 16 digits of precision and printed with up to 16 digits of precision.

A single precision constant is any numeric constant that has one of the following properties:

1. Seven or fewer digits
2. Exponential form denoted by E
3. A trailing exclamation point (!)

A double precision constant is any numeric constant that has one of the following properties:

1. Eight or more digits
2. Exponential form denoted by D
3. A trailing number sign (#)

The following are examples of constants:

Single Precision	Double Precision
46.8	345692811
−1.09E−06	−1.09432D−06
3489.0	3489.0#
22.5!	7654321.1234

Numeric constants in Microsoft BASIC cannot contain commas.

2.9 Variables

Variables represent values that are used in a program. As with constants, there are two types of variables: numeric and string. A numeric variable can only be assigned a value that is a number. A string variable can only be assigned a character string value. The value of the variable can be assigned by the user, or it can be assigned as the result of calculations in the program. In either case, the variable must always match the type of data that is assigned to it.

Before a variable is assigned a value, its value is assumed to be zero (numeric variables) or null (string variables).

2.9.1 Variable Names

A Microsoft BASIC variable name can contain as many as 255 characters. Only the first 40 characters are significant. The characters allowed in a variable name are letters, numbers, and the decimal point. The first character in a variable name must be a letter. Special type declaration characters are also allowed (see Section 2.9.2).

A variable name cannot be a reserved word, but embedded reserved words are allowed. If a variable begins with FN, it is assumed to be a call to a user-defined function. (See "DEF FN," in Section 3.11, for more information on user-defined functions.)

A variable name cannot be a reserved word. For example,

```
10 LOG = 8
```

is illegal because LOG is a reserved word. Reserved words include all Microsoft BASIC commands, statements, function names, and operator names (see Appendix I).

2.9.2 Declaring Variable Types

Variable names can declare either a numeric value or a string value.

String variable names are written with a dollar sign ($) as the last character. For example:

```
A$ = "SALES REPORT."
```

The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string.

Numeric variable names can declare integer, or single precision, or double precision values. Computations with integer and single

precision variables are less accurate than those with double precision variables. However, you may want to declare a variable to a lower precision type because:

1. Variables of higher precision take up more memory space.
2. Arithmetic computation times are longer for higher precision numbers. A program with repeated calculations runs faster with integer variables.

The type declaration characters for numeric variables and the memory requirements (in bytes) for storing each variable type are as follows.

Table 2.4. Variable Types

Declaration Character	Variable Type	Bytes Required
%	Integer	2
!	Single precision	4
#	Double precision	8
$	String	3 bytes overhead plus the present contents of the string

The default type for a numeric variable is single precision.

Examples of Microsoft BASIC variable names:

PI#	Declares a double precision value
MINIMUM!	Declares a single precision value
LIMIT%	Declares an integer value
N$	Declares a string value
ABC	Represents a single precision value

There is a second method by which variable types can be declared. The Microsoft BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL can be included in a program to declare the types for certain variable names. These statements are described in detail in Section 3.12,"DEFINT/SNG/DBL/STR."

2.9.3 Array Variables

An array is a group or table of values referenced by the same variable name. The individual values in an array are called elements. Array elements are variables also. They can be used in any BASIC statement or function which uses variables. Declaring the name and type of an array and setting the number of elements in the array is known as *dimensioning* the array.

Each array element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. Note that the array variable T and the variable T are not the same variable. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

Array elements, like numeric variables, require a certain amount of memory space, depending on the variable type. The memory requirements for storing arrays are as follows.

Element Type	Bytes
Integer	two per element
Single Precision	four per element
Double Precision	eight per element

2.9.4 ERR and ERL Variables

When an error-handling routine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain the line number 65535. To test if an error occurred in a direct statement, use:

```
IF 65535 = ERL THEN ...
```

Otherwise, use:

```
IF ERR = error code THEN ...
IF ERL = line number THEN ...
```

If the line number is not given on the right side of the relational operator, it will not be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. Microsoft BASIC error codes are listed in Appendix J.

2.10 Type Conversion

When necessary, Microsoft BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the numeric constant will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a TYPE MISMATCH error occurs.) Example:

```
10 A% = 23.42
20 PRINT A
RUN
 23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision. Example:

```
10 D = 6 /7
20 PRINT D
RUN
 .8571428571428571
```

The arithmetic was performed in double precision and the result was returned in D as a double precision value.

Note

Both operands must be double precision variables. If one of the variables is a single precision variable, then the last eight digits in the result are meaningless. Example:

```
10 D = 6 /7
20 PRINT D
RUN
 .857143
```

The arithmetic was performed in double precision and the result was returned to D (a single precision variable), rounded and printed as a single precision value.

---

3. Logical operators (see Section 2.11.3) convert their operands to integers and return an integer result. Operands must be in the range −32768 to +32767 or an OVERFLOW error occurs.

4. When a floating-point value is converted to an integer, the fractional portion is rounded. Example:

```
10 C% = 55.88
20 PRINT C%
RUN
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits rounded of the converted number will be valid, since only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E−8 times the original single precision value. Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04      2.039999961853027
```

2.11 Expressions and Operators

An expression can be a string or numeric constant, a variable, or a single value obtained by combining a constant and a variable with an operator.

Operators perform mathematical or logical operations on values. The operators provided by Microsoft BASIC can be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

2.11.1 Arithmetic Operators

The Microsoft BASIC arithmetic operators, in order of operational precedence, are listed in Table 2.5.

Table 2.5. Microsoft BASIC Arithmetic Operators

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
−	Negation	−X
*	Multiplication	X*Y
/	Floating-point division	X/Y
+,−	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operation is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
$X+2Y$	X+Y*2
$X - \frac{Y}{Z}$	X−Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
X(−Y)	X*(−Y)

Note

Two consecutive operators must be separated by parentheses.

Integer Division and Modulo Arithmetic

Two additional operations are available in Microsoft BASIC: integer division and modulo arithmetic.

Integer Division

Integer division is denoted by the backslash (\ ) instead of a ÷ sign. The operands are rounded to integers (must be in the range −32768 to +32767) before the division is performed, and the quotient is truncated to an integer. For example:

```
10 X = 10\4
20 Y = 25.68\6.99
30 PRINT X;Y
RUN
 2        3
```

Integer division follows multiplication and floating-point division in the established order of operational precedence.

Modulo Arithmetic

Modulo arithmetic is denoted by the operator MOD. Modulo arithmetic provides the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2          (10 4=2 with a remainder 2)
25.68 MOD 6.99 = 5      (26 7=3 with a remainder 5)
```

Modulo arithmetic immediately follows integer division in the established order of operational precedence.

Overflow and Division by Zero

If during the evaluation of an expression a division by zero is encountered, the DIVISION BY ZERO error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the DIVISION BY ZERO error message is displayed, positive machine infinity (the highest number the computer can produce) is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the OVERFLOW error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

2.11.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (−1) or "false" (0). This result can then be used to make a decision regarding program flow (see Section 3.30, "IF...THEN[...ELSE]").

Table 2.6 lists the relational operators.

Table 2.6. Relational Operators

Operator	Relation Tested	Expression
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See Section 3.35, "LET.")

When arithmetic and relational operators are combined in one expression, the arithmetic operation is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z.

2.11.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used to make a decision (see Section 3.30, "IF...THEN[...ELSE]"). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

A logical operator returns a result from the combination of true-false operands. The result (in bits) is either "true" (not zero) or "false" (zero). The true-false combinations and the results of a logical operation are known as *truth tables.*

There are six logical operators in Microsoft BASIC. They are: NOT (logical complement), AND (conjunction), OR (disjunction), XOR (exclusive or), IMP (implication), and EQV (equivalence). Each operator returns results as indicated in Table 2.7. A "1" indicates a true value and a "0" indicates a false value. Operators are listed in order of precedence.

Table 2.7. Logical Truth Tables

Operation	Values		Results
NOT	X		NOT X
	1		0
	0		1
AND	X	Y	X AND Y
	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	X	Y	X OR Y
	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	X	Y	X XOR Y
	1	1	0
	1	0	1
	0	1	1
	0	0	0
IMP	X	Y	X IMP Y
	1	1	1
	1	0	0
	0	1	1
	0	0	1
EQV	X	Y	X EQV Y
	1	1	1
	1	0	0
	0	1	0
	0	0	1

In an expression, logical operations are performed after arithmetic and relational operations.

Logical operators convert their operands to 16-bit, signed, two's complement integers in the range −32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or −1, logical operators return 0 or −1, respectively. The given operation is performed on these integers in bits, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator can be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator can be used to "merge" two bytes to create a particular binary value. The following examples demonstrate how the logical operators work.

63 AND 16 = 16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16.
15 AND 14 = 14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110).
−1 AND 8 = 8	−1 = binary 1111111111111111 and 8 = binary 1000, so −1 AND 8 = 8.
4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110).
10 OR 10 = 10	10 = binary 1010, so 1010 OR 1010 = 1010 (10).
−1 OR −2 = −1	−1 = binary 1111111111111111 and −2 = binary 1111111111111110, so −1 OR −2 = −1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of −1.
NOT X = −(X + 1)	The two's complement of any integer is the bit complement plus one.

2.11.4 Functional Operators

A function is used in an expression to call a predetermined operation to be performed on an operand. Microsoft BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). Microsoft BASIC functions are described in Chapter 4.

You can also define your own functions (known as "user-defined") with the DEF FN statement (see Section 3.11).

2.11.5 String Operators

A string expression is an expression that contains string constant(s) or string variable(s), or a combination of both (with operators) that evaluates to a single value.

There are two classes of string operations: concatenation and string function.

Concatenation

Combining two strings together is called concatenation. The plus symbol (+) is the concatenation operator. For example,

```
10 A$ = "FILE" : B$ = "NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$

RUN

FILENAME
  NEW FILENAME
```

combines the string variables A$ and B$ to produce the value "FILENAME."

String Function

Strings can be compared using the same relational operators that are used with numbers:

```
=    <>    <    >    <=    >=
```

A string function is the same as a numeric function, except the result is a string value. String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If during string comparison the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant.

Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL" > "CL"
"kg"> "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"                    (where B$ = "8/12/78")
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

2.12 Input Editing

If an incorrect character is entered as a line is being typed, it can be deleted with CONTROL-H or the ← (backspace) key. Both keys backspace over a character and erase it. Once a character has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type CONTROL-U or the → (retype) key. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC will automatically replace the old line with the new line.

To delete the entire program that currently resides in memory, enter the NEW command (see Chapter 3).

Microsoft BASIC has other sophisticated editing facilities that are part of the EDIT command. EDIT is discussed in Section 3.16.

2.13 Error Messages

If BASIC detects an error that terminates program execution, an error message is printed. For a complete list of Microsoft BASIC error codes and error messages, see Appendix J.

Chapter 3

Microsoft BASIC Commands and Statements

Chapter 3

Microsoft BASIC Commands and Statements

Microsoft BASIC commands and statements are described in this chapter. Each description has the following components:

Syntax	Shows the correct syntax for the instruction.
Purpose	Tells what the instruction is used for.
Remarks	Describes in detail how the instruction is used.
Example	Shows sample programs or program segments that demonstrate the use of the instruction.

Syntax notation for all commands and statements is given in Chapter 1. Numeric and string arguments (where applicable) have been abbreviated as follows:

X and Y	Represent any numeric expressions.
I and J	Represent integer expressions.
X$ and Y$	Represent string expressions.

If a floating-point value is supplied where an integer is required, BASIC will round the fractional portion and use the resulting integer.

3.1 AUTO

Syntax AUTO [<line number> [,<increment>]]

Purpose To generate a line number automatically after every carriage return.

Remarks AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing CONTROL-C. The line in which CONTROL-C is typed is not saved. After CONTROL-C is typed, BASIC returns to command level.

Examples

AUTO 100,50	Generates line numbers 100, 150, 200...
AUTO	Generates line numbers 10, 20, 30, 40....

3.2 BEEP

Syntax BEEP <pitch> <duration>

Purpose To create a tone of specified pitch and duration.

Remarks <pitch> is the desired pitch or frequency of a tone. Zero (0) is the highest pitch; 255 is the lowest.

<duration> is the desired duration in clock cycles. Zero (0) is the shortest duration; 255 is the maximum duration. A duration of 255 lasts approximately one second.

BEEP is intended for sound effect purposes. No attempt has been made to match <pitches> or <durations> with specific musical notes or note lengths.

Example

```
10 BEEP PDL(0), PDL(1):GOTO 10
```

3.3 CALL

Syntax 1

CALL <variable name>[(<argument list>)]

Purpose

To call an assembly language subroutine.

Remarks

The CALL statement allows you to transfer program flow to an external subroutine. This can also be done with the USR function. (See "USR," Section 4.44.)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> cannot be an array variable name.

<argument list> contains the arguments that are passed to the external subroutine. <argument list> can contain only variables.

Example

```
110 MYROUT = &HD000
120 CALL MYROUT(I,J,K)
 .
 .
 .
```

3.4 CHAIN

Syntax CHAIN [MERGE] < filespec>[,[<line number exp>] [,ALL][,DELETE<m – n>]]

Purpose To call a program and pass variables to it from the current program.

Remarks <filespec> contains the name of the program called. For example:

```
CHAIN"A:PROG1.BAS"
```

or

```
:CHAIN"B:SECPROG6.BAS"
```

The first example calls the BASIC program PROG1 from disk drive A:. The second example calls the BASIC program SECPROG6 from drive B:. If no other options are included by the user, CHAIN will load the called program and execute it beginning at the first line.

The MERGE option of the CHAIN statement merges the called overlay into the currently running program. That is, the program lines of the overlay are inserted into the current program in sequential order, beginning at the point specified by <line number exp>. The called program must be an ASCII file if it is to be merged. Example 1 in this section shows how the MERGE option is used.

Note

The CHAIN statement when used with the MERGE option leaves the files open and preserves the current OPTION BASE setting.

When using the MERGE option, insert user-defined functions before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions are left undefined after the merge operation is complete.

If you choose not to use MERGE, CHAIN will clear the effect of ON ERROR GOTO, disallow program continuation, reset all DATA pointers, and close all files. User-defined functions are preserved only if the corresponding DEF FN statements are not altered by MERGE. CHAIN without MERGE does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

<line number exp> is the line number (or an expression that evaluates to a line number) in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.

Note

<line number exp> is not affected by a RENUM command.

ALL is an option that allows all variables to pass from the current program to the overlay. If the ALL option is used, every variable in the current program is passed to the overlay. If you do not use ALL, a COMMON statement must be used to pass variables to the overlay. With COMMON, you can specify the variables to be passed. Array variables can be used by appending parentheses to the variable list. Note that the same variable cannot appear in more than one COMMON statement.

DELETE <m-n> is the option that deletes a range of lines in the original program after the overlay has been executed. m is the beginning line number and n is the last line number of the overlay range.

Example 1

```
10 REM THIS PROGRAM DEMONSTRATES CHAIN-
ING USING COMMON TO PASS
20 REM VARIABLES. SAVE THIS MODULE ON DISK
AS "PROG1" AND USE THE A OPTION.
30 DIM A$(2),B$(2)
40 COMMON A$(), B$()
50 A$(1) = "VARIABLES IN COMMON MUST BE
ASSIGNED"
60 A$(2) = "VALUES BEFORE CHAINING."
70 B$(1) = "": B$(2) = ""
80 CHAIN "PROG2"
90 PRINT: PRINT B$(1): PRINT: PRINT B$(2):
PRINT
100 END
```

```
10 REM THE STATEMENT "DIM A$(2),B$(2)" MAY
ONLY BE EXECUTED
20 REM ONCE. HENCE, IT DOES NOT APPEAR IN
THIS MODULE. SAVE
30 REM THIS MODULE ON THE DISK AS "PROG2"
AND USE THE A OPTION>
40 COMMON A$(), B$()
50 PRINT: PRINT A$(1);A$(2)
60 B$(1) = "NOTE HOW THE OPTION OF SPECIFY-
ING A STARTING LINE NUMBER"
70 B$(2) = "WHEN CHAINING AVOIDS THE DIMEN-
SION STATEMENT IN 'PROG1'."
80 CHAIN "PROG1",90
90 END
```

Example 2

```
10 REM THIS PROGRAM DEMONSTRATES CHAIN-
ING USING THE MERGE AND ALL
20 REM OPTIONS. SAVE THIS MODULE ON THE DISK
AS "MAINPRG".
30 A$ = "MAINPRG"
40 CHAIN MERGE "OVERLAY1",1010,ALL
50 END
```

```
1000 REM SAVE THIS MODULE ON THE DISK AS
"OVERLAY1" USING THE A OPTION.
1010 PRINT A$;"HAS CHAINED TO OVERLAY."
1020 A$ = "OVERLAY1"
1030 B$ = "OVERLAY2"
1040 CHAIN MERGE "OVERLAY2", 1010, ALL, DELETE
1000-1050
1050 END
```

```
1000 REM SAVE THIS MODULE ON THE DISK AS
"OVERLAY 2" USING THE A OPTION.
1010 PRINT A$; "HAS CHANGED TO "; B$:"."
1020 END
```

3.5 CLEAR

Syntax CLEAR [,[<expression1>][,<expression2>]]

Purpose To set all numeric variables to zero and all string variables to null, and to close all open files; and, optionally, to set the end of memory and the amount of stack space.

Remarks <expression1> is a memory location which, if specified, sets the highest location available for use by BASIC.

<expression2> sets aside stack space for BASIC. The default is 512 bytes or one-eighth of the available memory, whichever is smaller.

BASIC allocates string space dynamically. An OUT OF STRING SPACE error occurs only if there is no free memory left for BASIC to use.

Examples

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000
```

3.6 CLOSE

Syntax CLOSE[[#]<file number>[,[#]<file number...>]]

Purpose To conclude I/O to a disk file. CLOSE can be used either as a command or as a statement.

Remarks <file number> is the number under which the file was opened. A CLOSE with no arguments closes all open files. The option tells CP/M that a file is to be opened.

The association between a particular file and file number terminates upon execution of a CLOSE statement. The file can then be reopened using the same or a different file number; likewise, that file number can now be reused to OPEN any file.

A CLOSE statement for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Examples

```
CLOSE #1

CLOSE 2, 3
```

3.7 COLOR

Syntax COLOR = <color number>

Purpose To set the color for plotting in low-resolution graphics mode.

Remarks The <color number> argument is an integer in the range 0 to 15. The default value is 0. The colors available and their corresponding numbers are:

0 black	8 brown
1 magenta	9 orange
2 dark blue	10 gray
3 purple	11 pink
4 dark green	12 green
5 gray	13 yellow
6 medium blue	14 aqua
7 light blue	15 white

COLOR specifies the color of the line or points for plotting. The GR statement specifies the background color of the screen. GR is normally set to zero (black). GR and COLOR cannot be the same color. (See Section 3.26, "GR.")

To find out the COLOR of a given point on the screen, use the SCRN function. (See Section 4.34, "SCRN.")

Important

The COLOR statement can be used in low-resolution graphics mode only.

Example

```
10 GR
20 COLOR = 13
```

3.8 COMMON

Syntax COMMON <list of variables>

Purpose To pass variables to a chained program.

Remarks The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements can appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

<list of variables> can include any variable type, including array variables.

Example

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
 .
 .
 .
```

3.9 CONT

Syntax CONT

Purpose To continue program execution after a CONTROL-C has been typed or a STOP or END statement has been executed.

Remarks Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values can be examined and changed using direct mode statements. Execution can be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT can be used to continue execution after an error occurs.

CONT is invalid if the program has been edited after the break has occurred.

Example

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
30.7692
Ok
CONT
115.9
```

3.10 DATA

Syntax DATA <list of constants>

Purpose To store the numeric and string constants that are accessed by the program's READ statement(s).

Remarks DATA statements are nonexecutable and can be placed anywhere in the program. A DATA statement can contain as many constants as will fit on a line (separated by commas). Any number of DATA statements can be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein can be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> can contain numeric constants in any format, i.e. fixed-point, floating-point, or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements can be reread from the beginning by using the RESTORE statement.

Example 1

```
.
.
.
80 FOR I = 1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2

```
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
 CITY            STATE           ZIP
 DENVER,         COLORADO        80211
```

This program reads string and numeric data from the DATA statement in line 30.

3.11 DEF FN

Syntax DEF FN<name>[(<parameter list>)] = <function definition>

Purpose To define and name a function written by the user.

Remarks <name> is any legal variable name. <name> must be preceded by DEF FN, and becomes the name of the function.

<parameter list> is composed of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

The variables in the <parameter list> represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

<function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name.

A variable name used in a <function definition> may or may not appear in the <parameter list>. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

User-defined functions can be numeric or string functions. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function and the argument type does not match, a TYPE MISMATCH error occurs.

A DEF FN statement must be executed before the function it defines can be called. If a function is called before it has been defined, an UNDEFINED USER FUNCTION error occurs. DEF FN is illegal in direct mode.

Example

```
.
.
410 DEF FNAB(X,Y) = X^3/Y^2
420 T = FNAB(I,J)
.
.
```

Line 410 defines the function FNAB. The function is called in line 420.

3.12 DEFINT/SNG/DBL/STR

Syntax DEF<type> <range(s) of letters>

Purpose To declare variables as integer, single precision, double precision, or string variable types.

Remarks <type> is a variable type (INT, SNG, DBL, or STR) and <range of letters> is the variable name or names.

A DEF statement declares that the variable names beginning with the letter(s) specified will assume that variable type. However, a type declaration character always takes precedence over a DEF statement in the typing of a variable.

If no type declaration statements are encountered, BASIC assumes that all variables without declaration characters are single precision variables.

Examples

10 DEFDBL L-P	All variables beginning with the letters L, M, N, O, and P will be double precision variables.
10 DEFSTR A	All variables beginning with the letter A will be string variables.
10 DEFINT I-N,W-Z	All variables beginning with the letters I, J, K, L, M, N, W, X, Y, and Z will be integer variables.

3.13 DEF USR

Syntax

DEF USR[<digit>]=<integer expression>

Purpose

To specify the starting address of an assembly language subroutine.

Remarks

<digit> can be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed.

The value of <integer expression> is the starting address of the USR routine (see Appendix E, "Microsoft BASIC Assembly Language Subroutines").

Any number of DEF USR statements can appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example

```
.
.
.
200 DEF USR0=24000
210 X=USR0(Y^2/2.89)
.
.
.
```

3.14 DELETE

Syntax DELETE[<line number>][– <line number>]

Purpose To delete program lines.

Remarks BASIC always returns to command level after a DELETE statement is executed. If <line number> does not exist, an ILLEGAL FUNCTION CALL error occurs.

Examples

DELETE 40	Deletes line 40.
DELETE 40 – 100	Deletes lines 40 through 100, inclusive.
DELETE – 40	Deletes all lines up to and including line 40.

3.15 DIM

Syntax DIM <list of subscripted variables>

Purpose To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a SUBSCRIPT OUT OF RANGE error occurs. The minimum value for a subscript is always zero, unless otherwise specified with the OPTION BASE statement. (See Section 3.51, "OPTION BASE.")

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
 .
 .
 .
```

3.16 EDIT

Syntax EDIT <line number>

Purpose To enter edit mode at a specified line.

Remarks In edit mode, it is possible to edit portions of a line without retyping the entire line. Upon entering edit mode, BASIC types the line number of the line to be edited, then types a space and waits for an edit mode subcommand.

Edit mode subcommands

Edit mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the edit mode subcommands can be preceded by an integer which causes the command to be executed that number of times. When an integer is not specified, it is assumed to be 1.

Edit mode subcommands can be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and restarting edit mode
7. Entering edit mode from a syntax error

Note

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, *i* represents an optional integer (the default is 1), and $ represents the <ESCAPE> key.

Moving the Cursor

Space Bar — Use the space bar to move the cursor to the right. *i* Space bar moves the cursor *i* spaces to the right. Characters are printed as you space over them.

CONTROL-H — In edit mode, *i* CONTROL-H moves the cursor *i* spaces to the left. Characters are printed as you backspace over them.

← — ← or backspace moves the cursor to the left. The cursor moves over the characters already printed, but does not delete them.

Inserting Text

I — I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the screen. To terminate insertion, type <ESCAPE>. If <RETURN> is typed during an insert command, the effect is the same as typing <ESCAPE> and then <RETURN>. During an insert command, CONTROL-H, ←, or the underscore key can be used to delete characters to the left of the cursor. CONTROL-H will move the cursor over the characters as you backspace over them. <SHIFT>, CONTROL-\, and underscore (when pressed simultaneously) will print an underscore for each character you delete. If an attempt is made to insert a character that will make the line longer than 255 characters, an audio beep (CONTROL-G) sounds and the character is not printed.

X — The X subcommand is used to extend the line. X moves the cursor to the end of the line, enters insert submode, and allows insertion of text as if an insert command had been given. When you are finished extending the line, type <ESCAPE> or <RETURN>.

Deleting Text

D — *i*D deletes *i* characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than *i* characters to the right of the cursor, *i*D deletes the remainder of the line.

H — H deletes all characters to the right of the cursor and then automatically enters insert submode. H is useful for replacing statements at the end of a line.

Finding Text

S — The subcommand *i*S<ch> searches for the *i*th occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K — The subcommand *i*K<ch> is similar to *i*S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

Replacing Text

C — The subcommand C<ch> changes the next character to <ch>. If you wish to change the next *i* characters, use the subcommand *i*C, followed by *i* characters. After the *i*th new character is typed, you exit the change submode and return to edit mode.

Ending and Restarting Edit Mode

<RETURN> — Typing <RETURN> prints the remainder of the line, saves the changes you made, and exits edit mode.

E — The E subcommand has the same effect as <RETURN>, except the remainder of the line is not printed.

Q — The Q subcommand returns to BASIC command level without saving any of the changes that were made to the line during edit mode.

L — The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in edit mode. L is usually used to list the line when first entering edit mode.

A — The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

CONTROL-A

To enter edit mode on the line you are currently typing, type CONTROL-A. BASIC responds with a carriage return, an exclamation point (!), and a space. The cursor will be positioned at the first character in the line. Proceed by typing an edit mode subcommand.

Note

If BASIC receives an unrecognizable command or illegal character while in edit mode, it sounds a beep (CONTROL-G) and the command or character is ignored.

Entering Edit Mode from a Syntax Error

When a syntax error is encountered during execution of a program, BASIC automatically enters edit mode at the line that caused the error. For example:

```
10 K = 2(4)
run
?Syntax error in
 10
```

When you finish editing the line and press <RETURN> (or the E subcommand), BASIC reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, first exit edit mode with the Q subcommand. BASIC will return to command level, and all variable values will be preserved.

Note

If you have just entered a line and wish to go back and edit it, the command "EDIT ." will enter edit mode at the current line. (The line number symbol "." always refers to the current line.)

3.17 END

Syntax END

Purpose To terminate program execution, close all files, and return to command level.

Remarks END statements can be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END statement is executed.

Example

```
520 IF K>1000 THEN END ELSE GOTO 20
```

3.18 ERASE

Syntax ERASE <array variable> [<array variable>...]

Purpose To eliminate arrays from a program.

Remarks Arrays can be redimensioned after they are erased, or the previously allocated array space in memory can be used for other purposes. If an attempt is made to redimension an array without first erasing it, a REDIMENSIONED ARRAY error occurs.

Example

```
10 DIM B(5)
 .
 .
 .
450 ERASE A,B
460 DIM B(99)
 .
 .
 .
```

3.19 ERROR

Syntax ERROR <integer expression>

Purpose 1) To simulate the occurrence of a BASIC error; or 2) to allow error codes to be defined by the user.

Remarks <integer expression> must be a value between 0 and 255. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix J), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used for the Microsoft BASIC error codes. (It is preferable to use the highest available values, so compatibility can be maintained when more error codes are added to Microsoft BASIC.) This user-defined error code can then be conveniently handled in an error-handling routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error-handling routine causes an error message to be printed and execution to halt.

Example 1

```
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
 String too long in line 30
```

Or, in direct mode:

```
ERROR 15          (You type this line.)
 String too long  (BASIC types this line.)
```

Example 2

```
.
.
.
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
 .
 .
 .
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS
$5000"
410 IF ERL = 130 THEN RESUME 120
 .
 .
 .
```

Note

Refer to Section 2.9.4 for more information on ERR and ERL.

3.20 FIELD

Syntax FIELD[#]<file number>, <field width> AS <string variable>...

Purpose To allocate space for variables in a random access file buffer.

Remarks A FIELD statement must be executed to get data out of a random access buffer after a GET statement has been executed or to enter data before a PUT statement is executed.

The FIELD statement contains three arguments. <file number> is the number under which the file was opened. <field width> is the number of characters to be allocated to <string variable>. <string variable> is a string variable that will be used for random file access.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened (see Section 3.50, "OPEN"). Otherwise, a FIELD OVERFLOW error occurs. (The default record length is 128.)

Any number of FIELD statements can be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Important

You cannot use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to the correct place in the random access file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

Example 1

```
FIELD 1,20 AS N$, 10 AS ID$, 40 AS ADD$
```

Allocates the first 20 positions (bytes) in the random access file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does not place any data in the random access file buffer. (See Section 3.42, "LSET and RSET," and Section 3.23, "GET.")

Example 2

```
10 OPEN "R,"#1,"A:PHONELST",35
15 FIELD #1, 2 AS RECNBR$,33 AS DUMMY$
20 FIELD #1, 25 AS NAME$, 10 AS PHONENBR$
25 GET #1
30 TOTAL = CVI(RECNBR)$
35 FOR I = 2 TO TOTAL
40 GET #1, I
45 PRINT NAME$, PHONENBR$
50 NEXT I
```

Example 2 illustrates a multiple defined FIELD statement. In line 15, the 35-byte field is defined for the first record to keep track of the number of records in the file. In the next loop of statements (35-50), line 20 defines the field for individual names and phone numbers.

Example 3

```
10 FOR LOOP% = 0 TO 7
20 FIELD #1, (LOOP%*16) AS OFFSET$, 16 AS
A$(LOOP%)
30 NEXT LOOP%
```

Example 3 shows the construction of a FIELD statement using an array of elements of equal size. The result is equivalent to the single declaration:

```
FIELD #1,16 AS A$(0), 16 AS A$(1),....,16 AS A$(6), 16
AS A$(7)
```

Example 4

```
5 NUMB% = 5
10 DIM SIZE% (NUMB%): REM ARRAY OF FIELD
SIZES
20 FOR LOOP% = 0 TO NUMB%: READ SIZE%
30 DATA 9, 10, 12, 21, 41
120 DIM A$(NUMB%): REM ARRAY OF FIELDED
VARIABLES
130 OFFSET% = 0
140 FOR LOOP% = 0 TO NUMB%
150 FIELD #1, OFFSET% AS OFFSET$, SIZE%
(LOOP%) AS A$(LOOP%)
160 OFFSET% = OFFSET% + SIZE%(LOOP%)
170 NEXT LOOP%
```

Example 4 creates a field in the same manner that Example 3 does. However, the element size varies with each element. The equivalent declaration is:

```
FIELD #1, SIZE%(0) AS A$(0), SIZE%(1) AS A$(1) ....
SIZE%(NUMB%) AS A$(NUMB%),
```

3.21 FILES

Syntax FILES [<filespec>]

Purpose To print the names of files residing on the current disk.

Remarks <filespec> is a string formula which can contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

If a disk drive is specified as part of the <filespec>, then files under the specified filename in that disk drive are listed. Otherwise, the current or default drive is used.

Examples
```
FILES
FILES "*.BAS"
FILES "B:*.*"
FILES "TEST?.BAS"
```

3.22 FOR...NEXT

Syntax
```
FOR <variable> = <x> TO <y> [STEP <z>]
 .
 .
 .
NEXT [<variable>] [,<variable>...]
```

Purpose To allow a series of instructions to be performed in a loop a given number of times.

Remarks <variable> is used as a counter and <x>, <y>, and <z> are numeric expressions. The first numeric expression, <x>, is the initial value of the counter. The second numeric expression, <y>, is the final value of the counter.

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value of <y>. If it is not greater, BASIC branches back to the statement which follows the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops can be nested; that is, a FOR...NEXT loop can be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement can be used for all of them.

The variable(s) in the NEXT statement can be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a NEXT WITHOUT FOR error message is issued and execution is terminated.

Example 1

```
10 K = 10
20 FOR I = 1 TO K STEP 2
30 PRINT I;
40 K = K + 10
50 PRINT K
60 NEXT
RUN
 1   20
 3   30
 5   40
 7   50
 9   60
```

Example 2

```
10 J = 0
20 FOR I = 1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3

```
10 I = 5
20 FOR I = 1 TO I + 5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

3.23 GET

Syntax 1 GET [#]<file number>[,<record number>]

Syntax 2 GET <keyboard character>

Purpose To read a record from a random access disk file into a random access buffer; or (in Syntax 2) to read a single character from the keyboard.

Remarks <file number> is the number under which the file was opened and <record number> is the number of the record to be read. The range is 1 to 32767.

If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

In Syntax 2, the <keyboard character> read from the keyboard is not displayed on the screen. It is not necessary to press the <RETURN> key. If CONTROL-@ is the keyboard character, it returns the ASCII null character. The result of getting a left-arrow or CONTROL-H can also print as if the null character were being returned.

Note

After a GET statement has been executed, INPUT# and LINE INPUT# can be executed to read characters from the random access file buffer.

Example

```
10 OPEN "R",#1,"B: VENDOR, 85
20 FIELD #1, 20 AS VENDNAME$, 30 AS ADDR$,
30 GET #1
35 AS CITY$
40 PRINT VENDNAME$, ADDR$, CITY$
```

3.24 GOSUB...RETURN

Syntax

```
GOSUB <line number>
.
.
.
RETURN
```

Purpose

To branch to and return from a subroutine.

Remarks

<line number> is the first line of the subroutine.

A subroutine can be called any number of times in a program, and a subroutine can be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine can contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines can appear anywhere in the program, but it is recommended that subroutines be readily distinguishable from the main program.

To prevent inadvertent entry into the subroutine, the GOSUB statement can be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

To prevent stack overflow, a subroutine called by a GOSUB statement must always exit through a RETURN statement.

Note

You can use an ON...GOSUB statement to branch to different subroutines based on the result of an expression.

Example

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT "IN";
60 PRINT "PROGRESS"
70 RETURN

RUN
 SUBROUTINE IN PROGRESS
 BACK FROM SUBROUTINE
```

3.25 GOTO

Syntax

GOTO <line number>

Purpose

To branch unconditionally out of the normal program sequence to a specified line number.

Remarks

If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example

```
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
 R = 5          AREA = 78.5
 R = 7          AREA = 153.86
 R = 12         AREA = 452.16
?Out of data in 10
```

3.26 GR

Syntax GR <screen number> [,<color number>]

Purpose To initialize low-resolution graphics mode and set the background color of the screen.

Remarks <screen number> is an integer in the range 0-1 and <color number> is an integer in the range 0-15.

<screen number> specifies the mode to be used as follows:

Screen Number	Mode
0	40x40 graphics plus 4 lines text (mixed text display mode)
1	40x48 graphics with no lines text (low-resolution graphics display mode)

If <screen number> is not specified, a default value of zero is assumed.

GR clears the screen when it initializes low-resolution graphics mode.

<color number> specifies the background color of the screen. The <color number> argument is

optional. If <color number> is not specified, color is set to black. The GR <color number> argument cannot be the same as the COLOR <color number>. The color names and their associated numbers are:

0 black	8 brown
1 magenta	9 orange
2 dark blue	10 gray
3 purple	11 pink
4 dark green	12 green
5 gray	13 yellow
6 medium blue	14 aqua
7 light blue	15 white

Examples

GR	Same as Applesoft GR statement.
GR 1,15	Set the screen background to white and initialize the low-resolution screen display mode (40x48).

3.27 HLIN

Syntax HLIN <x1 coordinate>,<x2 coordinate>AT <y coordinate>

Purpose To draw a horizontal line from point (x1,y) to point (x2,y) (in low-resolution graphics display mode only).

Remarks x1 and x2 are integers in the range 0 to 39 and y is an integer in the range 0 to 47. The <x1 coordinate> must be less than or equal to the <x2 coordinate>.

The color of the line is specified by the most recently executed COLOR statement.

If any of the coordinates are not in the required range as specified above, an ILLEGAL FUNCTION CALL error results.

The HLIN statement normally draws a line composed of dots from x1 to x2 at the vertical coordinate y. However, if used when in text display mode, or when in mixed graphics display mode with y in the range 40 to 47, a line of characters is displayed instead of the line of dots.

Example

```
10 GR
20 COLOR=3
30 HLIN 14,20 AT 39
```

3.28 HOME

Syntax HOME

Purpose To clear the screen of all text and move the cursor to the upper left corner of the screen.

Remarks When HOME is used with an external terminal, it sends a "clear screen" character sequence to the terminal. HOME can only be used with terminals that support this feature.

Example

```
10 HOME
20 VTAB 12
30 PRINT "A CLEAN SCREEN"
```

3.29 HTAB

Syntax HTAB <screen position number>

Purpose To move the cursor to the screen position that is <screen position number> spaces from the left edge of the current screen line.

Remarks The first (leftmost) position on the line is 1, the last (rightmost) position on the line is 40.

HTAB uses absolute moves, not relative moves. For instance, if the cursor was at position 10, and the command HTAB 13 was executed, the cursor would be moved to position 13, not position 23.

If a <screen position number> greater than 40 but less than 255 is specified, it will be treated modulo 40. The command HTAB 60 would place the cursor at position 20 on the current line. A <screen position number> greater than 255 results in an ILLEGAL FUNCTION CALL error.

3.30 IF...THEN[...ELSE] and IF...GOTO

Syntax IF <expression> THEN <clause> [ELSE <clause>]

Syntax IF <expression> GOTO <line number> [ELSE <clause>]

Purpose To make a decision regarding program flow based on the result returned by an expression.

Remarks

<expression> is a unique expression which sets the conditions for the IF statement to make a decision about which program path to follow. <clause> can be a BASIC statement or statements, or a line number to branch to.

If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN can be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero (false), the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

Nesting of IF Statements

IF...THEN[...ELSE] statements can be nested. Nesting is limited only by the length of the line. For example,

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
THEN PRINT "LESS THAN" ELSE PRINT
"EQUAL"
```

is a legal statement. If a statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example,

```
IF A=B THEN IF B=C THEN PRINT "A=C"
ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in direct mode, an UNDEFINED LINE error results, unless a statement with the specified line number had previously been entered in indirect mode.

Note

When using IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A – 1.0)< 1.0E – 6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E−6.

Example 1

```
200 IF I THEN GET#1,I
```

This statement GETs record number I, if I is not zero.

Example 2

```
100 IF(I<20)*(I>10) THEN DB = 1979 – 1:GOTO 300
110 PRINT "OUT OF RANGE"
 .
 .
 .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the screen or to the line printer, depending on the value of the variable (IOFLAG). If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the screen.

3.31 INPUT

Syntax INPUT[;][<"prompt string">;]<variable list>

Purpose To allow input from the keyboard during program execution.

Remarks When an INPUT statement is encountered, program execution pauses and a question mark is displayed to indicate the program is waiting for data.

If INPUT is immediately followed by a semicolon, then the <RETURN> typed by the user to input data does not echo a carriage return/linefeed sequence.

If <"prompt string"> is included, the string is displayed before the question mark. The required data is then entered at the keyboard. A comma can be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will display the prompt with no question mark.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list can be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the message ?REDO FROM START to be printed. No assignment of input values is made until an acceptable response is given.

Examples

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
? 5                (The 5 was typed in by the user in
                    response to the question mark)
5 SQUARED IS 25

10 PI = 3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A = PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4      (User types 7.4)

THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
```

3.32 INPUT#

Syntax INPUT#<file number>,<variable list>

Purpose To read data items from a sequential disk file and assign them to program variables.

Remarks <file number> is the number used when the file was opened for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.)

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. Unlike INPUT, no question mark is printed with INPUT#.

With numeric values, leading spaces, carriage returns, and linefeeds are ignored. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a number. The number terminates on a space, carriage return, linefeed, or comma.

If BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns, and linefeeds are also ignored. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string cannot contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string. It will terminate on a comma, carriage return or linefeed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
 EBENEEZER SCROOGE
 SUPER MANN
 Input past end in 20
```

3.33 INVERSE

Syntax INVERSE

Purpose To set video output so that the screen displays dark characters on a light background.

Remarks When using an external terminal, INVERSE sends a highlighted character sequence to terminals that support this feature.

INVERSE does not affect characters that are already on the screen when INVERSE is executed.

The NORMAL command restores the usual light letters on a dark background. (See Section 3.47, "NORMAL.")

Example

```
10 PRINT "THESE ARE WHITE CHARACTERS"
20 INVERSE
30 PRINT "THESE ARE BLACK CHARACTERS"
40 NORMAL
```

3.34 KILL

Syntax KILL <filespec>

Purpose To delete a file from disk.

Remarks KILL is used for all types of disk files: program files, random access data files, and sequential data files.

If a KILL statement is given for a file that is currently OPEN, a FILE ALREADY OPEN error occurs.

Example

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$ = "" THEN 200 'CARRIAGE RETURN EXITS
INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR = 53 AND ERL = 20 THEN OPEN
"O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

See also Appendix D, "Microsoft BASIC Disk I/O."

3.35 LET

Syntax [LET] <variable> = <expression>

Purpose To assign the value of an expression to a variable.

Remarks Notice the word LET is optional; i.e., the equal sign is all that is required to assign an expression to a variable name.

Attempting to assign a numeric value to a string variable or a string value to a numeric variable will result in a TYPE MISMATCH error.

Examples

```
110 LET D = 12
120 LET E = 12^2
130 LET F = 12^4
140 LET SUM = D + E + F
 .
 .
 .
```

or

```
110 D = 12
120 E = 12^2
130 F = 12^4
140 SUM = D + E + F
 .
 .
 .
```

3.36 LINE INPUT

Syntax LINE INPUT[;][<"prompt string">;] <string variable>

Purpose To input an entire line (up to 254 characters) to a string variable without the use of delimiters.

Remarks The optional <"prompt string"> is a string literal that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string.

<string variable> is the input. All input from the end of the prompt to the <RETURN> is assigned to <string variable>. However, if a linefeed/carriage return sequence (this order only) is encountered, both characters are echoed, but the carriage return is ignored, the linefeed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/linefeed sequence at the keyboard.

A LINE INPUT statement can be escaped from by typing CONTROL-C. BASIC will return to BASIC command level and displays the "Ok" prompt. Typing CONT resumes execution at the LINE INPUT statement.

Example See the example in the following section (LINE INPUT#).

3.37 LINE INPUT#

Syntax LINE INPUT#<file number>,<string variable>

Purpose To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks <file number> is the number under which the file was opened and <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a <RETURN>. It then skips over the linefeed/carriage return sequence. The next LINE INPUT# reads all characters up to the next <RETURN>. (If a linefeed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

Example

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
 CUSTOMER INFORMATION? LINDA JONES  234,4
 MEMPHIS
 LINDA JONES  234,4  MEMPHIS
```

3.38 LIST

Syntax 1 LIST [<line number>]

Syntax 2 LIST [<line number>[–[<line number>]]]

Purpose To list all or part of the program currently in memory on the screen.

Remarks BASIC always returns to command level after LIST is executed.

Syntax 1

If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either when the end of the program is reached or by typing CONTROL-C.) If <line number> is included, only the specified line is listed. CONTROL-S suspends a listing. Pressing CONTROL-S again (or CONTROL-Q or any other key) allows the listing to continue.

Syntax 2

This format allows the following options:

1. If only the first number is specified, that line and all subsequent lines are listed.
2. If only the second number is specified, all lines from the beginning of the program through that line are listed.
3. If both numbers are specified, the entire range is listed.

Examples

Syntax Format 1

LIST	Lists the program currently in memory.
LIST 500	Lists line 500.

Syntax Format 2

LIST 150 –	Lists all lines from 150 to the end.
LIST – 1000	Lists all lines from the lowest number through 1000.
LIST 150 – 1000	Lists lines 150 through 1000, inclusive.

3.39 LLIST

Syntax LLIST [<line number>[–[<line number>]]]

Purpose To list all or part of the program currently in memory to the line printer.

Remarks BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Syntax 2.

<line number> is a valid line number in the range 0 to 65529.

LLIST assumes a 132-character-wide printer.

Examples

LLIST 150–	Lists all lines from 150 to the end.
LLIST –1000	Lists all lines from the lowest number through 1000.
LLIST 150–1000	Lists lines 150 through 1000, inclusive.

3.40 LOAD

Syntax LOAD <filespec>[,R]

Purpose To load a file from disk into memory.

Remarks <filespec> includes the name and extension of the file saved. With CP/M, the default extension .BAS is supplied.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the R option is used with LOAD, the program is RUN after it is loaded, and all open data files are kept open. Thus, LOAD with the R option can be used to chain several programs (or segments of the same program). Information can be passed between the programs using disk data files.

Examples

```
LOAD "STRTRK",R

LOAD "B:MYPROG"
```

3.41 LPRINT and LPRINT USING

Syntax LPRINT [<list of expressions>]

Syntax LPRINT USING <string exp>;<list of expressions>

Purpose To print data at the line printer.

Remarks Same as PRINT and PRINT USING (Sections 3.55 and 3.56), except output goes to the line printer.

LPRINT assumes a 132-character-wide printer.

3.42 LSET and RSET

Syntax LSET <string variable> = <string expression>

Syntax RSET <string variable> = <string expression>

Purpose To move data from memory to a random access file buffer (in preparation for a PUT statement).

Remarks If <string expression> requires fewer bytes than were fielded to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See Section 4.27, "MKI$, MKS$, MKD$."

Note

LSET or RSET can also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$ = SPACE$(20)
120 RSET A$ = N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

Examples

```
150 LSET A$ = MKS$(AMT)
160 LSET D$ = DESC$
```

See also Program 6 in Appendix D, "Microsoft BASIC Disk I/O."

3.43 MERGE

Syntax MERGE <filespec>

Purpose To merge a specified ASCII disk file into the program currently in memory.

Remarks <filespec> is the filename and extension of the saved file. CP/M will append a default filename extension of .BAS if one was not supplied in the SAVE command. Refer to Section 2.4, "CP/M File Naming Conventions" for more information about possible filename extensions under CP/M. The file must be saved in ASCII format. (If not, a BAD FILE MODE error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (Merging may be

thought of as "inserting" the program lines on disk into the program in memory.)

BASIC always returns to command level after a MERGE operation.

Examples

```
MERGE "A:CATPRO"

MERGE "B:LSTPROG"
```

See also Example 1 for the CHAIN statement in this chapter.

3.44 MID$

Syntax

MID$(<string exp1>,n[,m]) = <string exp2>

Purpose

To replace a portion of one string with another string.

Remarks

n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

MID$ is also a function that returns a substring of a given string (see Section 4.26).

Example

```
10 A$ = "KANSAS CITY, MO"
20 MID$(A$,14) = "KS"
30 PRINT A$
Ok
RUN
 KANSAS CITY, KS
```

3.45 NAME

Syntax NAME <filespec> AS <new filename>

Purpose To change the name of a disk file.

Remarks <filespec> is a file specification as outlined under "CP/M File Naming Conventions" in Section 2.4. <new filename> is the new filename. It must be a valid filename as outlined in the same section.

<filespec> must exist and <new filename> must not exist; otherwise, an error will result. If the device name is omitted, the current drive is assumed. After NAME is executed, the file exists on the same disk, in the same area of disk space, under the new name.

Example NAME "A:ACCTS" AS "LEDGER"

In this example, the disk file that was formerly named ACCTS in drive A: will now be named LEDGER.

3.46 NEW

Syntax NEW

Purpose To delete the program currently in memory and clear all variables.

Remarks NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW command is executed.

3.47 NORMAL

Syntax NORMAL

Purpose To restore video output to the normal light characters on dark background.

Remarks NORMAL is used in conjunction with the INVERSE command. (See Section 3.33.)

NORMAL does not affect characters already displayed on the screen in INVERSE mode when the NORMAL command is executed.

For external terminals that support the highlight feature of INVERSE, NORMAL sends a "low-light" character sequence instead of a dot.

3.48 ON ERROR GOTO

Syntax ON ERROR GOTO <line number>

Purpose To enable error trapping and specify the first line of the error-handling routine.

Remarks Once error trapping has been enabled, all errors detected, including direct mode errors (e.g. , syntax errors), will generate a jump to the specified error-handling routine. If <line number> does not exist, an UNDEFINED LINE error results.

To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors generate an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error-handling routine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error-handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note

If an error occurs during execution of an error-handling routine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error-handling routine.

Example 10 ON ERROR GOTO 1000

3.49 ON...GOSUB and ON...GOTO

Syntax ON <expression> GOSUB <list of line numbers>

Syntax ON <expression> GOTO <list of line numbers>

Purpose To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON... GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an ILLEGAL FUNCTION CALL error occurs.

Example 100 ON L-1 GOTO 150,300,320,390

3.50 OPEN

Syntax OPEN <mode>,[#]<file number>,<filespec> [,<reclen>]

Purpose To allow I/O to a disk file.

Remarks <mode> is a string expression whose first character is one of the following:

O	specifies sequential output mode
I	specifies sequential input mode
R	specifies random access input/output mode

<file number> is an integer expression with a value between 1 and 15. The <file number> is associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filespec> is a string expression for a file specification which contains a name that conforms to CP/M's rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random access files. <reclen> is not valid for sequential files. The default record length is 128 bytes. To use OPEN with record lengths longer than 128 bytes, see Section 2.1,"Initialization."

A disk file must be opened before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

Note

A file can be opened for sequential input or random access on more than one file number at a time. A file can be opened for output, however, on only one file number at a time.

Example 10 OPEN "I",2,"INVEN"

See also the example for the FIELD statement in this chapter.

3.51 OPTION BASE

Syntax OPTION BASE <n>

Purpose To declare the minimum value for array subscripts.

Remarks n is either 1 or 0. The default value is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript can have is 1.

OPTION BASE must be coded before you define or use any arrays.

3.52 PLOT

Syntax PLOT <x coordinate>, <y coordinate>

Purpose To plot a dot on the screen in low-resolution graphics mode.

Remarks <x coordinate> is an integer in the range 0-39 and <y coordinate> is an integer in the range 0-47.

The coordinate points (0,0) are located in the upper left corner of the screen.

The color of the dot placed by PLOT is determined by the most recently executed COLOR or GR statement.

PLOT normally places a dot at (x,y). However, if PLOT is used while in text mode or in mixed text mode with the y coordinate in the range 40 to 47, a character is displayed instead of a dot.

If either <x coordinate> or <y coordinate> is not in the required range specified above, an ILLEGAL FUNCTION CALL error results.

Example

```
GR
COLOR = 9
PLOT 24,37
```

3.53 POKE

Syntax POKE I,J

Purpose To write a byte into a memory location.

Remarks I and J are integer expressions. The expression I represents the address of the memory location and J represents the data byte. I must be in the range −32768 to 65535. For interpretation of negative values of I, see Section 4.46,"VARPTR."

The complementary function of POKE is PEEK (see Section 4.30). The argument to PEEK is an address from which a byte is to be read.

POKE and PEEK are useful for efficiently storing data, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example

```
10 POKE 106,0
```

3.54 POP

Syntax POP

Purpose To return from a subroutine that was branched to by a GOSUB statement without branching back to the statement following the most recent GOSUB statement.

Remarks POP is used instead of a RETURN to nullify a GOSUB statement. Like RETURN, it nullifies the last GOSUB in effect, but it does not return to the statement following the GOSUB. After a POP, the

next RETURN encountered will branch to one statement beyond the second most recently executed GOSUB. Thus, POP, in effect, takes one address off the top of the "stack" of RETURN addresses.

See also "GOSUB...RETURN" in Section 3.24.

Example 10 POP 106,0

3.55 PRINT

Syntax PRINT [<list of expressions>]

Purpose To display data on the screen.

Remarks If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are displayed on the screen. The expressions in the list can be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a <RETURN> is printed at the end of the line. If the printed line is longer than the screen width, BASIC goes to the next physical line to continue printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 10^{-7} is output as .0000001 and 10^{-8} is output as 1E−08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format are output using the unscaled format. For example, 1D−15 is output as .0000000000000001 and 1D−16 is output as 1D−16.

A question mark can be used in place of the word PRINT in a PRINT statement.

Example 1

```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
 10          0          -25          3125
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
  9 SQUARED IS 81 AND 9 CUBED IS 729
? 21
  21 SQUARED IS 441 AND 21 CUBED IS 9261
?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line. Line 40 causes a blank line to be printed before the next prompt.

Example 3

```
10 FOR X = 1 TO 5
20 J = J + 5
30 K = K + 10
40 ?J;K;
50 NEXT X
Ok
RUN
 5   10   10   20   15   30   20   40   25   50
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and a positive number is preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

3.56 PRINT USING

Syntax PRINT USING <string exp>;<list of expressions>

Purpose To print strings or numbers using a specified format.

Remarks <string exp> is a string literal (or variable) composed of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

<list of expressions> consists of the string expressions or numeric expressions that are to be printed, separated by semicolons.

String Fields

When PRINT USING is used to print strings, one of three formatting characters can be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\"

Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$ = "LOOK":B$ = "OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\   \";A$;B$
50 PRINT USING" \   \";A$;B$;"!!"
RUN
 LO
 LOOKOUT
 LOOK OUT   !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input.

Example:

```
10 A$ = "LOOK":B$ = "OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
 LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters can be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by the appropriate number of spaces) in the field.

A decimal point can be inserted at any position in the field. If the format string specifies that a digit is to precede the

decimal point, the digit will always be printed (as 0, if necessary). Numbers are rounded, as necessary.

Examples:

```
PRINT USING "##. ##";.78
  0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.## ";
10.2,5.3,66.789,.234

10.20  5.30  66.79  0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

\+ A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.

\– A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.

Examples:

```
PRINT USING " + ##.##    "
; – 68.95,2.40,55.6, – .90

– 68.95   + 2.40   + 55.60   – 0.90

PRINT USING "##.## –    "
; – 68.95,22.449, – 7.01

68.95 –   22.45   7.01 –
```

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

Example:

```
PRINT USING"**#.#";12.39, -0.9,
765.1

*12.4  *-0.9  765.1
```

$$ — $$ specifies two more digit positions, one of which is the dollar sign. A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

Example:

```
PRINT USING "$$###.##";456.78
  $456.78
```

**$ — The **$ at the beginning of a format string combines the effects of the ** and $$ symbols (see above). Leading spaces are asterisk-filled and a dollar sign is printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

Example:

```
PRINT USING "**$##.##";2.34
***$2.34
```

A comma specifies another digit position. A comma to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma at the end of the format string is printed as part of the string. The comma has no effect if used with exponential (^^^^) format.

Examples:

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

^^^^

Four carets can be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position can be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or − sign is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

Examples:

```
PRINT USING "##.##^^^^";234.56
  2.35E+02

PRINT USING ".####^^^^-";888888
  .8889E+06

PRINT USING "+.##^^^^";123
 +.12E+03
```

_

An underscore in the format string causes the next character to be output as a literal character.

Example:

```
PRINT USING "__!##.##__!";12.34
!12.34!
```

The literal character itself may be an underscore by placing two underscore characters (__ __) in the format string.

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign is printed in front of the rounded number.

Examples:

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an ILLEGAL FUNCTION CALL error results.

3.57 PRINT# and PRINT# USING

Syntax PRINT#<filenumber>,[USING<string exp>;]<list of expressions>

Purpose To write data to a sequential disk file.

Remarks <filenumber> is the number used when the file is opened for output.

<string exp> is composed of formatting characters as described in the previous section (PRINT USING).

The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as

it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data on the disk so that it will be input correctly from the file.

In the list of expressions, numeric expressions should be delimited by semicolons. For example:

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the file.)

String expressions must be separated by semicolons in the list. To format string expressions correctly on the file, use explicit delimiters in the list of expressions.

Examples Let A$="CAMERA" and B$="93604−1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604−1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement, as follows:

```
PRINT#1,A$;",";B$
```

The image written to the file is

```
CAMERA,93604 – 1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or linefeeds, write them to the file surrounded by explicit quotation marks using CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604−1". The statement

```
PRINT#1,A$;B$
```

would write the following image to the file:

```
CAMERA, AUTOMATIC   93604 − 1
```

The statement

```
INPUT#1,A$,B$
```

inputs "CAMERA" to A$ and "AUTOMATIC 93604−1" to B$. To separate these strings properly on the file, write double quotation marks to the file image using CHR$(34).The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;
CHR$(34)
```

writes the following image to the file:

```
"CAMERA, AUTOMATIC""   93604 − 1"
```

The statement

```
INPUT#1,A$,B$
```

inputs "CAMERA, AUTOMATIC" to A$ and " 93604−1" to B$.

The PRINT# statement can also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

For more examples using PRINT#, see the example for the KILL statement in this chapter and Program 1 in Appendix D.

3.58 PUT

Syntax PUT [#]<file number>[,<record number>]

Purpose To write a record from a random access buffer to a random access disk file.

Remarks <file number> is the number under which the file was opened and <record number> is the record number for the record to be written.

If <record number> is omitted, the record will assume the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

Note

PRINT#, PRINT# USING, and WRITE# can be used to put characters in the random access file buffer before a PUT statement is executed.

In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a FIELD OVERFLOW error.

Example See the examples in Appendix D.

3.59 RANDOMIZE

Syntax RANDOMIZE [<expression>]

Purpose To reseed the random number generator.

Remarks

The optional <expression> argument is a numeric expression. If <expression> is omitted, BASIC suspends program execution and asks for a value by printing

```
Random Number Seed (-32768 to 32767)?
```

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
Ok
RUN
 RANDOM NUMBER SEED (-32768 to 32767)? 3
```

You type 3:

```
 .88598   .484668   .586328   .119426   .709225
RUN
 RANDOM NUMBER SEED (-32768 to 32767)? 4
```

You type 4 for new sequence:

```
 .803506   .162462   .929364   .292443   .322921
RUN
 RANDOM NUMBER SEED (-32768 to 32767)? 3
```

Same sequence as first RUN:

```
 .88598   .484668   .586328   .119426   .709225
```

3.60 READ

Syntax READ <variable list>

Purpose To read values from a DATA statement and assign them to variables.

Remarks A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables can be numeric or string. The values read must agree with the variable types specified. If they do not agree, a SYNTAX ERROR will result.

A single READ statement can access one or more DATA statements (they will be accessed in order), or several READ statements can access the same DATA statement. If the number of variables in <variable list> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement.

Example 1

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2

```
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", "COLORADO","80211"
40 PRINT C$,S$,Z
Ok
RUN
 CITY          STATE         ZIP
 DENVER,       COLORADO      80211
```

This program reads string and numeric data from the DATA statement in line 30.

3.61 REM

Syntax REM <remark>

Purpose To allow explanatory remarks to be inserted in a program.

Remarks REM statements are not executed, but are output exactly as entered when the program is listed.

REM statements can be branched into or from a GOTO or GOSUB statement; execution continues with the first executable statement after the REM statement.

Remarks can be added to the end of a line by preceding the <remark> with a single quotation mark instead of REM. <remark> can consist of any sequence of characters.

Warning

Do not use REM in a data statement, as it is considered legal data.

Examples

```
.
.
.
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I = 1 TO 20
140 SUM = SUM + V(I)
.
.
.
```

or:

```
.
.
.
120 FOR I = 1 TO 20     'CALCULATE AVERAGE
VELOCITY
130 SUM = SUM + V(I)
140 NEXT I
.
.
.
```

3.62 RENUM

Format RENUM [[<new number>][,[<old number>][,<increment>]]]

Purpose To renumber program lines.

Remarks The <new number> argument is the first line number to be used in the new sequence. The default new number is 10.

The <old number> argument is the line in the current program where renumbering is to begin. The default old number is the first line of the program.

The <increment> argument is the increment to be used in the new sequence. The default value is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message UNDEFINED LINE xxxxx IN yyyyy is printed. The incorrect line number reference xxxxx is not changed by RENUM, but line number yyyyy can be changed.

Note

RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20, and 30) or to create line numbers greater than 65529. If it is used for such purposes, an ILLEGAL FUNCTION CALL error occurs.

Examples

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will be numbered in increments of 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will be numbered in increments of 50.
RENUM 1000, 900,20	Renumbers the lines from 900 up so they start with line number 1000 and continue in increments of 20.

3.63 RESET

Syntax RESET

Purpose To execute a disk system reset. After disks are exchanged in a disk drive, RESET must be executed before reading or writing to the new disk. This is called a "warm start."

Remarks Always execute a RESET command after changing disks. Otherwise, you will not be able to write to the new disks.

RESET also closes all open files. Therefore, when changing disks, a CLOSE statement should be executed before removing the old disks.

3.64 RESTORE

Syntax RESTORE [<line number>]

Purpose To allow DATA statements to be reread beginning at a specified line.

Remarks After RESTORE is executed, the next READ statement accesses the first item in the first DATA statement in the program. The optional <line number> argument specifies a DATA statement. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
 .
 .
 .
```

3.65 RESUME

Syntax RESUME [{NEXT | <0> | <line number>}]

Purpose To continue program execution after an error recovery procedure has been performed.

Remarks Any of the arguments shown above can be used, depending upon where execution is to resume:

RESUME
or
RESUME 0

Execution resumes at the statement that caused the error.

RESUME NEXT

Execution resumes at the statement immediately following the one that caused the error.

RESUME <line number>

Execution resumes at <line number>.

A RESUME statement that is outside an error-handling routine generates a RESUME WITHOUT ERROR message.

Example

```
10 ON ERROR GOTO 900
 .
 .
 .
900 IF (ERR = 230)AND(ERL = 90) THEN PRINT "TRY
AGAIN":RESUME 80
 .
 .
 .
```

3.66 RUN

Syntax 1 RUN [<line number>]

Purpose To execute the program currently in memory.

Remarks If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC always returns to command level after a RUN is executed.

Example

```
RUN
```

Syntax 2 RUN <filespec>[,R]

Purpose To load a file from disk into memory and run it.

Remarks <filespec> is a string expression that includes the name used when the file was saved. With CP/M, if no filename extension is given the default extension .BAS is supplied.

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the R option, all data files remain OPEN.

Examples

```
RUN "NEWFIL",R

RUN "B:PROG"
```

See also the programs listed in Appendix D, "Microsoft BASIC Disk I/O."

3.67 SAVE

Syntax SAVE <filespec>[,{<A>|<P>}]

Purpose To save a program file on disk.

Remarks <filespec> is a string expression that includes the name used when the file was saved. With CP/M, if no filename extension is given, the default extension .BAS is supplied. If the<filespec> already exists, the file will be overwritten.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For example, the MERGE command requires an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or loaded), any attempt to list or edit it will fail.

Warning

Once the P option is used, a file cannot be "unprotected."

Examples

```
SAVE"COM2",A

SAVE"PROG",P

SAVE "B:PROG"
```

See also the programs listed in Appendix D, "Microsoft BASIC Disk I/O."

3.68 STOP

Syntax STOP

Purpose To terminate program execution and return to command level.

Remarks STOP statements can be used to terminate execution anywhere in a program. When a STOP is encountered, the following message is printed:

```
BREAK IN LINE nnnnn
```

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command.

Example

```
10 INPUT A,B,C
20 K = A^2*5.3:L = B^3/.26
30 STOP
40 M = C*K + 100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
```

3.69 SWAP

Syntax SWAP <variable>,<variable>

Purpose To exchange the values of two variables.

Remarks Any type variable can be swapped (integer, single precision, double precision, string), but the two swapped variables must be of the same type, or a TYPE MISMATCH error results.

Example

```
10 A$ = " ONE " : B$ = " ALL " : C$ = "FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
```

3.70 SYSTEM

Syntax SYSTEM

Purpose To close all files and return to CP/M command level.

Remarks You cannot use CONTROL-C to return to CP/M, as it always returns you to BASIC.

Example

```
SYSTEM
A>
```

3.71 TEXT

Syntax TEXT

Purpose To reset the screen to normal full Apple text display mode (24x40) from low-resolution graphics or high-resolution graphics display modes.

Remarks TEXT will clear the screen if it is used to return from low-resolution graphics. It will not clear the screen if used to return from high-resolution graphics.

If used while in text display mode, TEXT has the same effect as VTAB 24.

Example

```
10 GR
20 COLOR = 5
30 VLIN 24,30 AT 35
40 TEXT
50 PRINT "THIS IS A VERTICAL LINE"
```

3.72 TRACE/NOTRACE

Syntax TRACE

Syntax NOTRACE

Purpose To trace the execution of program statements.

Remarks The TRACE statement (executed in either direct or indirect mode) is an aid to troubleshooting which enables a trace flag that prints each line number of the program as it is executed. The numbers are enclosed in square brackets where they are displayed. The trace flag can be disabled either with the NOTRACE statement or when a NEW command is executed.

Example

```
10 K = 10
20 FOR J = 1 TO 2
30 L = K  +  10
40 PRINT J;K;L
50 K = K + 10
60 NEXT
70 END
TRACE
Ok
RUN
 [10][20][30][40] 1 10 20
 [50][60][30][40] 2 20 30
 [50][60][70]
Ok
NOTRACE
```

3.73 VLIN

Syntax

VLIN <y1 coordinate>, <y2 coordinate> AT <x coordinate>

Purpose

To draw a vertical line from the point at (x,y1) to the point at (x,y2) on the screen in low-resolution graphics mode only.

Remarks

The <y1 coordinate> and <y2 coordinate> are integers in the range 0-47. The <x coordinate> is an integer in the range 0-39. The <y1 coordinate> must be less than or equal to the <y2 coordinate>.

If any of the coordinates are not in the required range as specified above, an ILLEGAL FUNCTION CALL error results.

The color of the line is determined by the most recent COLOR statement.

The VLIN statement normally draws a line from coordinates y1 to y2 at the horizontal coordinate x. However, if used when in text mode or when in

mixed graphics mode with y2 in the range 40 to 47, the part of the line that falls in the text area will be displayed as a line of characters.

Example

```
10 GR
20 COLOR = 3
30 VLIN 20,45 AT 12
```

3.74 VTAB

Syntax VTAB <screen line number>

Purpose To move the cursor vertically to the line on the screen that corresponds to the specified <screen line number>.

Remarks The first line (the top line) on the screen is line 1; the last line (the bottom line) on the screen is line 24.

VTAB uses absolute moves. For instance, if the cursor was on line 10 of the screen and the command VTAB 13 was executed, the cursor would be moved to line 13, not line 23.

If a <screen line number> greater than 24 is specified, it will be treated as modulo 24. For example, the command VTAB 26 would place the cursor on screen line 2. If a <screen line number> greater than 255 is specified, it results in an ILLEGAL FUNCTION CALL error.

VTAB can move the cursor either up or down.

When used with an external terminal, VTAB sends a "cursor address" character sequence to terminals that address this feature.

Example

```
10 VTAB 12: PRINT "MIDDLE OF SCREEN"
```

3.75 WAIT

Syntax WAIT <address>, I[,J]

Purpose To suspend program execution while you monitor the status of an address.

Remarks I and J are integer expressions.

The WAIT statement causes execution to be suspended until a specified address develops a specified bit pattern. The data read at the port performs an exclusive OR operation with the integer expression J, and then performs an AND operation with I. If the result of either is zero, BASIC loops back and reads the data at the address again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

Warning

It is possible to enter an infinite loop with the WAIT statement. If this happens, you must manually restart the computer. To avoid this and continue execution, WAIT must have the specified value (I or J) at <address> at some point in the program execution.

Example

```
100 WAIT &HE000,128
200 PRINT "KEYPRESS!":GOTO 100
```

3.76 WHILE...WEND

Syntax

```
WHILE <expression>
 .
 .
[<loop statements>]
 .
 .
WEND
```

Purpose

To execute a series of statements in a loop as long as a given condition is true.

Remarks

If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops can be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a WHILE WITHOUT WEND error, and an unmatched WEND statement causes a WEND WITHOUT WHILE error.

Example

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS = 1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115         FLIPS = 0
120         FOR I = 1 TO J-1
130                 IF A$(I) > A$(I + 1) THEN
                            SWAP A$
                                (I),A$(I + 1):FLIPS = 1
140         NEXT I
150 WEND
```

3.77 WIDTH

Syntax WIDTH [LPRINT] <integer expression>

Purpose To set the line width for the screen or line printer to a specified number of characters.

Remarks <integer expression> must have a value in the range 15 to 255. The default width is 80 characters.

If the LPRINT option is omitted, the line width is set at the screen. If LPRINT is included,the line width is set at the line printer.

If <integer expression> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given in the POS or LPOS function, returns to zero after position 255.

Example

```
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
 ABCDEFGHIJKLMNOPQR
 STUVWXYZ
```

3.78 WRITE

Syntax WRITE [<expression>,<expression>,...]

Purpose To output data on the screen.

Remarks If <expression> is omitted, a blank line is output. If <expression> is included, the values of the expression(s) are output on the screen. The expressions can be numeric and/or string expressions. They must be separated by commas.

In printed output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/linefeed sequence.

WRITE outputs numeric values using the same format as the PRINT statement.

Example

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80, 90,"THAT'S ALL"
```

3.79 WRITE#

Syntax

WRITE#<file number>,<expression>,
[<expression>,...]

Purpose

To write data to a sequential file.

Remarks

<file number> is the number under which the file was opened in "O" mode. The <expression>s can be either string or numeric expressions. They must be separated by commas.

WRITE#, unlike PRINT#, inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/linefeed sequence is inserted after the last item in the list is written to disk.

Example

Let A$="CAMERA" and B$="93604−1". The statement

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604 – 1"
```

A subsequent INPUT# statement, such as

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604−1" to B$.

Chapter 4
Microsoft BASIC Functions

Chapter 4
Microsoft BASIC Functions

The intrinsic functions provided by Microsoft BASIC are described in this chapter. The functions can be called from any program without further definition.

Functions differ from commands and statements in that they cannot be performed by themselves. They must be used in conjunction with either a statement or a command. If used with an assignment statement (=), a function must appear on the right side of the = sign.

Each function description consists of the following components:

Syntax	Shows the correct format for the function.
Action	Describes the action the function takes.
Remarks	Describes in detail how the function is used; also discusses special conditions for using the function.
Example	Shows sample programs or program segments that demonstrate the use of the instruction.

Syntax notation for all functions is given in Chapter 1. Numeric and string arguments (where applicable) have been abbreviated as follows:

X and Y	Represent any numeric expressions.
I and J	Represent integer expressions.
X$ and Y$	Represent string expressions.

If a floating-point value is supplied where an integer is required, BASIC rounds the fractional portion and uses the resulting integer.

4.1 ABS

Syntax ABS(X)

Action Returns the absolute value of the expression X.

Example
```
PRINT ABS(7*(-5))
 35
```

4.2 ASC

Syntax ASC(X$)

Action Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix H for ASCII codes.)

Remarks If X$ is null, an ILLEGAL FUNCTION CALL error is returned.

See Section 4.6, "CHR$" for ASCII-to-string conversion.

Example
```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
```

4.3 ATN

Syntax ATN(X)

Action Returns the arctangent of X in radians.

Remarks The result is in the range −pi/2 to pi/2.

The calculation of ATN(X) is performed in single precision format, regardless of the declared variable type (integer, single precision, or double precision) of X.

Example

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
```

4.4 BUTTON

Syntax BUTTON(I)

Action Returns the current value of the push button on the Apple game controller specified by I.

Remarks I is in the range 0 to 3.

The returned value is either 0 if the button is not currently depressed, or −1 if the button is currently depressed.

Example

```
10 IF BUTTON(0) THEN PRINT "BOOM"
```

4.5 CDBL

Syntax CDBL(X)

Action Converts X to a double precision number.

Example

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67     454.6700134277344
```

4.6 CHR$

Syntax CHR$(I)

Action Returns a single element whose ASCII string is code I. (ASCII codes are listed in Appendix H.)

Remarks CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character (CHR$(7)) could be sent as a preface to an error message.

See Section 4.2, "ASC" for ASCII-to-numeric conversion.

Example

```
PRINT CHR$(66)
 B
```

4.7 CINT

Syntax CINT(X)

Action Converts X to an integer.

Remarks Converts X to an integer by rounding the fractional portion. If X is not in the range −32768 to +32767, an OVERFLOW error occurs.

See Section 4.5, "CDBL", and Section 4.9, "CSNG," for converting numbers to double precision and single precision data types. See also Section 4.13, "FIX," and Section 4.19, "INT," both of which return integers.

Example

```
PRINT CINT(45.67)
 46
```

4.8 COS

Syntax COS(X)

Action Returns the cosine of X.

Remarks COS is the trigonometric cosine function. X must be in radians. To convert from degrees to radians, multiply by pi/180 (pi = 3.141593).

The calculation of COS(X) is performed in single precision, regardless of the declared variable type (integer, single precision, or double precision) of X.

Example

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
```

4.9 CSNG

Syntax CSNG(X)

Action Converts X to a single precision number.

Remarks See Section 4.7, "CINT," and Section 4.5,"CDBL," for converting numbers to the integer and double precision data types.

Example

```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421     975.342
```

4.10 CVI, CVS, CVD

Syntax

```
CVI(<2-byte string>)
CVS(<4-byte string>)
CVD(<8-byte string>)
```

Action Convert string variable values to numeric variable values.

Remarks Numeric values that are read in from a random access disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

See also Section 4.27, "MKI$, MKS$, MKD$," and Section D.3.2, "Random Access Files," in Appendix D.

Example

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y = CVS(N$)
.
.
.
```

4.11 EOF

Syntax EOF(<file number>)

Action Tests for an end-of-file condition.

Remarks <file number> is the number specified in the OPEN statement.

The EOF function returns −1 (true) if the end of a sequential file has been reached. Use EOF to test for an end-of-file condition while inputting, to avoid INPUT PAST END errors.

Example

```
10 OPEN "I",1,"DATA"
20 C = 0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C = C + 1:GOTO 30
.
.
.
```

4.12 EXP

Syntax EXP(X)

Action Calculates the exponential function *e*.

Remarks EXP returns the mathematical number *e* raised to the X power. X must be <=87.3365. If EXP overflows, the OVERFLOW error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

The calculation of EXP(X) is performed in single precision format, regardless of the declared variable type (integer, single precision, or double precision) of X.

Example

```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
```

4.13 FIX

Syntax FIX(X)

Action Truncates X to an integer.

Remarks FIX(X) is equivalent to the expression SGN(X)* INT (ABS(X)). The difference between FIX and INT is that FIX does not return the next lower number when X is negative, as INT does.

See Section 4.19, "INT," and Section 4.7, "CINT." They also return an integer.

Examples

```
PRINT FIX(58.75)
 58

PRINT FIX(-58.75)
-58
```

4.14 FRE

Syntax FRE{(0) | (<X$>)}

Action Returns the number of bytes in memory not being used by BASIC.

Remarks Strings in BASIC often have variable lengths. That is, each time you assign a value to a string, its length can change. Strings are also manipulated dynamically. For this reason, string space can be scattered or fragmented.

FRE(" ") forces a reallocation of memory space (otherwise known as "housecleaning," "garbage collection," etc.) before returning the number of free bytes. Housecleaning collects useful data and frees up unused areas of memory that were once used for strings. The data is compressed so you can use memory space more efficiently.

BASIC initiates housecleaning when all free memory is used up. The housecleaning process can take from a minute to a minute and a half.

Arguments to FRE are dummy arguments.

Example

```
PRINT FRE(0)
 14542
```

Note

The actual value returned by the FRE function may differ from the value returned in this example.

4.15 HEX$

Syntax HEX$(X)

Action Returns a string that represents the hexadecimal value of the decimal argument.

Remarks X is rounded to an integer before HEX$(X) is evaluated.

See Section 4.28, "OCT$," for octal conversion.

Example

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ "HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
```

4.16 INKEY$

Syntax INKEY$

Action Reads a character from the keyboard.

Remarks INKEY$ returns either a one-character string containing a character read from the keyboard; or a null string if no character is pending at the keyboard. No characters are echoed and all characters are passed through to the program except for CONTROL-C, which terminates the program.

Example

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$ = " "
1020 FOR I% = 1 TO TIME LIMIT%
1030 A$ = INKEY$ : IF LEN(A$) = 0 THEN 1060
1040 IF ASC(A$) = 13 THEN TIMEOUT% = 0 :
RETURN
1050 RESPONSE$ = RESPONSE$ + A$
1060 NEXT I%
1070 TIMEOUT% = 1 : RETURN
```

4.17 INPUT$

Syntax INPUT$(X[,[#]Y])

Action Returns a string of X characters, read from the keyboard or from file number Y.

Remarks X is a string of characters and Y is the file number used in the OPEN statement. File number 0 is used to denote the keyboard.

If the keyboard is used for input, no characters are echoed and all CONTROL characters are passed through; except CONTROL-C, which is used to interrupt the execution of the INPUT$ function.

Example 1

```
5 'LIST THE CONTENTS OF A SEQUENTIAL
FILE IN HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2

```
.
.
.
100 PRINT "TYPE P TO PROCEED OR S TO
STOP"
110 X$ = INPUT$(1)
120 IF X$ = "P" THEN 500
130 IF X$ = "S" THEN 700 ELSE 100
.
.
.
```

4.18 INSTR

Syntax INSTR([I,]X$,Y$)

Action Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the start position.

Remarks I is a numeric expression in the range 1 to 255.

If I > LEN(X$), or if X$ is null, or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ can be string variables, string expressions, or string literals.

If I=0 is specified, the error message ILLEGAL ARGUMENT IN <line number> is returned.

Example

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
```

4.19 INT

Syntax INT(X)

Action Returns the largest integer <=X.

Remarks See Section 4.7, "CINT," and Section 4.13, "FIX," which also return integer values.

Examples

```
PRINT INT(99.89)
 99

PRINT INT(-12.11)
 -13
```

4.20 LEFT$

Syntax LEFT$(X$,I)

Action Returns a string composed of the leftmost I characters of X$.

Remarks I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Also see Section 4.26, "MID$," and Section 4.32, "RIGHT$."

Example

```
10 A$ = "BASIC"
20 B$ = LEFT$(A$,5)
30 PRINT B$
RUN
 BASIC
```

4.21 LEN

Syntax LEN(X$)

Action Returns the number of characters in X$.

Remarks Nonprinting characters and blanks are counted.

Example

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
RUN
 16
```

4.22 LOC

Syntax LOC(<file number>)

Action Returns the current position in the file.

Remarks With random access disk files, LOC returns the record number just read or written from a GET or PUT statement. If the file was opened but no disk I/O has been performed yet, LOC returns a 0.

With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was opened.

Example

```
200 IF LOC(1)>50 THEN STOP
```

4.23 LOF

Syntax LOF(<file number>)

Action Returns the number of records present in the last extent (128 records) read or written. If the file does not exceed one extent, then LOF returns the true length of the file.

Example

```
110 IF NUM%>LOF(1) THEN PRINT "INVALID
ENTRY"
```

4.24 LOG

Syntax LOG(X)

Action Returns the natural logarithm of X.

Remarks X must be greater than zero.

The calculation of LOG(X) is performed in single precision, regardless of the declared variable type (integer, single precision, or double precision) of X.

Example

```
PRINT LOG(45/7)
 1.86075
```

4.25 LPOS

Syntax LPOS(X)

Action Returns the current position of the line printer print head within the line printer buffer.

Remarks LPOS does not necessarily give the physical position of the print head. X is a dummy argument.

Example

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

4.26 MID$

Syntax MID$(X$, I [,J])

Action Returns a string of length J characters from X$ beginning with the Ith character.

Remarks I and J must be in the range 1 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

If I=0 is specified, the error message ILLEGAL ARGUMENT IN <line number> is returned.

Also see Section 4.20, "LEFT$," and Section 4.32, "RIGHT$."

Example

```
10 A$ = "GOOD "
20 B$ = "MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
 GOOD EVENING
```

4.27 MKI$, MKS$, MKD$

Syntax

```
MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)
```

Action Convert numeric values to string values.

Remarks Any numeric value that is placed in a random access file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

See also Section 4.10, "CVI, CVS, CVD," and Section D.3.2, "Random Access Files," in Appendix D.

Example

```
90 AMT = (K + T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
 .
 .
```

4.28 OCT$

Syntax OCT$(X)

Action Returns a string which represents the octal value of the decimal argument.

Remarks X is rounded to an integer before OCT$(X) is evaluated.

See Section 4.15, "HEX$," for hexadecimal conversion.

Example

```
PRINT OCT$(24)
 30
```

4.29 PDL

Syntax PDL(I)

Action Returns the current value of the controller knob, in the range 0 to 255, of the game controller specified by I.

Remarks I is an integer in the range 0 to 3.

The values of two game controllers should not be read in consecutive instructions, as the reading from the first may affect the second. A delay such as 10 FOR X=1 TO 10: NEXT X between the two instructions provides sufficient separation for a correct reading.

Example

```
10 PRINT PDL(0): GOTO 10
RUN
 0
 23
 79
 100
 190
 255
 C
 BREAK IN
```

4.30 PEEK

Syntax PEEK(I)

Action Returns the byte read from the indicated memory location (I).

Remarks I must be in the range −32768 to +65535. The returned value is an integer in the range 0 to 255. For an interpretation of a negative value of I, see Section 4.46, "VARPTR."

PEEK is the complementary function of the POKE statement (see Section 3.53).

Example

```
A = PEEK(&H5A00)
```

4.31 POS

Syntax

```
POS(I)
```

Action Returns the current cursor position.

Remarks The current horizontal (column) position of the cursor is returned. The returned value is in the range of 1 (the leftmost position) to 80. X is a dummy argument.

Also see Section 4.25, "LPOS."

Example

```
IF POS(X)>60 THEN PRINT CHR$(13)
```

4.32 RIGHT$

Syntax

```
RIGHT$(X$,I)
```

Action Returns the rightmost I characters of string X$.

Remarks If I=LEN(X$), returns X$. If I=0, the null string (length zero) is returned.

Also see Section 4.26, "MID$," and Section 4.20, "LEFT$."

Example

```
10 A$ = "DISK BASIC"
20 PRINT RIGHT$(A$,8)
RUN
 BASIC
```

4.33 RND

Syntax RND[(X)]

Action Returns a random number between 0 and 1.

Remarks The same sequence of random numbers is generated each time the program is run, unless the random number generator is reseeded (see Section 3.59, "RANDOMIZE"). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example

```
10 FOR I = 1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24  30  31  51  5
```

4.34 SCRN

Syntax SCRN(X,Y)

Action Returns the code number of the color of the coordinate point specified by (X,Y).

Remarks X is an integer in the range 0 to 39 and Y is an integer in the range 0 to 47.

Example

```
10 GR
20 COLOR = 13
30 PLOT 10,15
40 PRINT SCRN(10,15)
RUN
 13
```

4.35 SGN

Syntax SGN(X)

Action Returns the mathematical sign (signum) function.

Remarks If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns −1.

Example

```
ON SGN(X)+2 GOTO 100,200,300
```

Branches to 100 if X is negative, 200 if X is 0, and 300 if X is positive.

4.36 SIN

Syntax SIN(X)

Action Calculates the trigonometric sine function of the angle X.

Remarks Returns the sine of X in radians.

The calculation of SIN(X) is performed in single precision format, regardless of the declared variable type (integer, single precision, or double precision) of X.

If you want to convert degrees to radians, multiply by pi/180 (pi = 3.141593).

Example

```
PRINT SIN(1.5)
 .997495
```

4.37 SPACE$

Syntax SPACE$(X)

Action Returns a string of spaces of length X.

Remarks The expression X is rounded to an integer and must be in the range 0 to 255.

Also see Section 4.38, "SPC."

Example

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
    2
      3
        4
          5
```

4.38 SPC

Syntax SPC(I)

Action Prints I spaces on the screen.

Remarks SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255. A semicolon (;) is assumed to follow the SPC(I) command.

Also see Section 4.37, "SPACE$."

Example

```
PRINT "OVER" SPC(15); "THERE"
 OVER                THERE
```

4.39 SQR

Syntax SQR(X)

Action Returns the square root of X.

Remarks X must be >=0.

Example

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.16228
 15            3.87298
 20            4.47214
 25            5
```

4.40 STR$

Syntax STR$(X)

Action Returns a string representation of the value of X.

Remarks The VAL function (Section 4.45) returns the inverse of the value of X.

Example

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
 .
 .
 .
```

4.41 STRING$

Syntax

```
STRING$(I,J)
STRING$(I,X$)
```

Action Returns a string of length I whose characters all have ASCII code J or the first character of X$.

Remarks I and J are in the range 0 to 255.

Example

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
```

4.42 TAB

Syntax

```
TAB(I)
```

Action Tabs to position I on the screen.

Remarks I must be in the range 1 to 255. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one.

TAB can only be used in PRINT and LPRINT statements.

Example

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
 NAME              AMOUNT
 G. T. JONES       $25.00
```

4.43 TAN

Syntax TAN(X)

Action Calculates the trigonometric tangent of the angle X.

Remarks Returns the tangent of X in radians.

The calculation of TAN(X) is performed in single precision, regardless of the declared variable type (integer, single precision, or double precision) of X.

If the result of a TAN operation overflows, the OVERFLOW error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example

```
10 Y = Q*TAN(X)/2
```

4.44 USR

Syntax USR[<digit>](X)

Action Calls the indicated assembly language subroutine with the argument X.

Remarks <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that subroutine. If <digit> is omitted, USR0 is assumed.

The CALL statement is another way to call an assembly language subroutine. See Appendix E for more information on using assembly language subroutines.

Example

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
 .
 .
 .
```

4.45 VAL

Syntax VAL(X$)

Action Returns the numerical value of string X$.

Remarks The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example,

```
VAL (" -3")
```

returns −3.

See also Section 4.40, "STR$," for numeric-to-string conversion.

Example

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
PRINT NAME$
TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815
THEN
PRINT
NAME$ TAB(25) "LONG BEACH"
.
.
.
```

4.46 VARPTR

Syntax VARPTR{<variable name> | #<file number>}

Action Returns the memory address of a variable or file control block.

Remarks For either argument, the returned address is an integer in the range 0 to 65535.

VARPTR(<variable name>) returns the address of the first byte of data identified with a variable. In the case of a string variable, VARPTR gets the address of the first byte of the string descriptor. (See Appendix E.2, "USR Function Calls.")

A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise, an ILLEGAL FUNCTION CALL error results. Any type of variable name can be used (numeric, string, array). The address returned is an integer in the range +32767 to −32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so the address can be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest addressed element of the array is returned.

Note

All simple variables should be assigned before calling VARPTR for an array, since the addresses of the arrays change whenever a new simple variable is assigned.

VARPTR(#<file number>) is used for sequential files. It returns the starting address of the disk I/O buffer assigned to <file number>. For random access files, it returns the address of the FIELD buffer assigned to <file number>.

Example

```
100 X = USR(VARPTR(Y))
```

4.47 VPOS

Syntax VPOS(X)

Action Returns the current vertical position of the cursor.

Remarks The topmost screen position is 1. X is a dummy argument.

Example

```
10 PRINT "NOW YOU SEE IT."
20 FOR T=0 TO 1000: NEXT T
30 VTAB VPOS(0) - 1
40 PRINT "NOW YOU DON'T"
```

Chapter 5
High-Resolution Graphics

Chapter 5
High-Resolution Graphics

The graphics commands in this chapter let you plot high-resolution shapes on a 280x160 or 280x192 screen grid. High-resolution graphics in the Microsoft BASIC Interpreter consists of the following commands:

HGR	For setting the high-resolution display mode
HCOLOR	For setting the background color of the screen
HPLOT	For plotting lines from point to point

The function HSCRN is also included as an aid in developing high-resolution graphics programs. High-resolution commands can also be used as statements in the indirect operational mode.

5.1 Differences Between High-Resolution and Low-Resolution Graphics

The high-resolution graphics mode is set within the HGR command. HGR allows either "mixed-text plotting" (graphics on a 280x160 grid in the upper part of the screen, and four lines of text in the lower part of the screen) or graphics plotting on a 280x192 grid (the entire screen).

The GR command sets the low-resolution display mode in mixed-text plotting on a 40x40 grid with 4 lines of text; or in graphics plotting on a 40x48 grid with no lines of text. The HGR command also differs from the GR command by two additional <screen number> options which can prevent the screen from being cleared when setting the display mode. This can be useful for programmers who want to draw a shape in their program then go back to text mode, but don't want the drawing erased while the program performs other functions.

5.1.1 The Use of Colors

The HGR and HCOLOR commands have a larger selection of black and white shades than the low-resolution GR and COLOR commands. The high-resolution commands also permit the use of inverse video when plotting. The low-resolution commands, however, give you a wider selection of colors.

5.1.2 Plotting High-Resolution Shapes

The syntax of the HPLOT commands allows the plotting of different shapes with the execution of a single command. For example, use HPLOT in the following statement to draw a triangle:

```
10 HPLOT 30,50 TO 35,40 TO 40,50 TO 30,50
```

To draw the same shape with low-resolution graphics, you would need to plot each individual point in the sloping sides of the triangle, as in the following sample of code:

```
10 HLINE 10,20 AT 25
20 PLOT 10,25
30 PLOT 11,24
40 PLOT 12,23
50 PLOT 13,22
60 PLOT 14,21
70 PLOT 15,20
80 PLOT 16,21
90 PLOT 17,22
100 PLOT 18,23
120 PLOT 19,24
130 PLOT 20,25
```

5.2 Sample Program

The following program demonstrates the use of high-resolution graphics commands in a program for black and white monitors.

```
1 REM GRFTST.BAS
2 DIM X(23),Y(23)
3 IF A = 1 THEN 200
10 HGR 1,3: HCOLOR = 0
20 HPLOT 140,96
30 FOR A = 0 TO 3.14159*20 STEP .05
40 R = SIN (A*2.9)
50 HPLOT TO 140 + 107*R*COS(A),96 + 95*R*SIN(A)
60 NEXT
70 HGR 1,12:FOR T = 0 TO 500:NEXT:HGR 1,12
80 GOTO 70
200 N = INT (RND*14) + 13
210 PI = 6.28318/N:FOR I = 0 TO N – 1:A = PI*I
220 X(I) = COS(A)*107 + 140:Y(I) = SIN(A)*95 + 96
230 NEXT
240 HGR 1,0:HCOLOR = 3
250 FOR I = 0 TO N – 1: FOR J – 1 TO N – 1:HPLOT X(I), Y(I) TO X(J),
Y(J):NEXT:NEXT
260 HGR 1,12:FOR T = 0 TO 500:NEXT:HGR 1,12
270 GOTO 260
```

5.3 HGR

Syntax HGR <screen number>, [<color number>]

Purpose Initializes high-resolution graphics mode.

Remarks

<screen number> is an integer in the range 0 to 3, and <color number> is an integer in the range 1 to 12. <screen number> specifies the screen grid to be used as follows:

Screen	Clear Screen	Screen Grid
0	yes	280x160 graphics and 4 lines of text
1	yes	280x192 graphics, no lines text
2	no	280x160 graphics and 4 lines text
3	no	280x192 graphics, no lines text

If <screen number> is not specified, <screen number>=0 is assumed.

The <color number> option specifies the color to be used and is optional. If <color number> is not specified, color is set to 1. When used with screen grids 0 and 1, <color number> will fill the screen with the color specified by <color number>. The following table lists the color names and their associated numbers.

0 black	5 orange	9 white1
1 green	6 blue	10 black2
2 violet	7 white	11 white2
3 white	8 black1	12 reverse
4 black		

Examples

```
10 HGR
```

This is the same as the Applesoft HGR statement.

10 HGR 1,2	Fills screen with violet, sets the 280x192 screen grid.
10 HGR 3	Sets 280x192 screen grid, doesn't clear screen.

Note

This statement can be used differently in GBASIC than it can in Applesoft.

5.4 HCOLOR

Syntax HCOLOR = <color number>

Purpose To set the color for plotting in high-resolution graphics mode.

Remarks <color number> is an integer in the range 0 to 12. The colors available and their corresponding numbers are:

0 black	5 orange	9 white1
1 green	6 blue	10 black2
2 violet	7 white	11 white2
3 white	8 black1	12 reverse
4 black		

To distinguish between the different shades of whites and blacks: Color codes 0, 3, 4, and 7 plot a very fine line. Colors black1, white1, black2, and white2 (8, 9, 10, and 11) plot a larger dot or thicker line that is equal in size (width) to dots or lines plotted with green, violet, orange, or blue. Black1 and white1 should be used with green or violet if you want dots or lines of the same position and width. Black2 and white2 should be used with orange or blue.

If you are using a black and white monitor, use 0, 3, 4, and 7.

<color number> can be specified in the HGR statement (see "HGR," Section 5.2.1). If it is not specified in HGR, it is set to zero by HGR until another color is specified with the HCOLOR statement.

HCOLOR can be used in high-resolution graphics mode only.

Note that because of the way in which home TVs work, a high-resolution dot plotted with HCOLOR=3 (white) or HCOLOR=7 (white) will be white only if both (x,y) and (x+1,y) are plotted. If only (x,y) is plotted, the dot will be blue when x is even and green when x is odd.

5.5 HPLOT

Syntax 1 HPLOT <x1>,<y1>[TO<x2>,<y2>...
[TO<xn>,<yn>]]

Purpose Plots a point or draws a line on the high-resolution screen, specified by points: (xl,y1), (x2,y2), etc.

Syntax 2 HPLOT TO <x2>, <y2>

Purpose Draws a line from the last dot plotted to the coordinate point at (x2,y2).

Remarks In Syntax 1, HPLOT <x1>, <y1> plots a single point. HPLOT <x1>, <y1> TO <x2>, <y2> TO ... <xn>, <yn> plots a line starting at (x1,y1) and proceeding through each of the points specified. The plotted line can be extended from point to point in the same HPLOT statement by specifying additional points, limited only by screen limits and the 239-character limit.

In Syntax 1, the color of the dot or line is determined by the most recent HCOLOR statement. If no color has been specified, the default color 0 will be assigned.

In Syntax 2, the color of the line is determined by the last HCOLOR executed. Syntax 2 cannot be used if no dot has previously been plotted.

Use HPLOT in high-resolution graphics mode only.

Example

```
10 HGR
20 HCOLOR = 2
30 HPLOT 24,125 TO 100,12 TO 270,1
```

5.6 HSCRN

Syntax HSCRN (X,Y)

Action In high-resolution graphics mode, HSCRN checks to see if a dot exists at a specified coordinate (X,Y). If a dot does exist, HSCRN returns −1 (true).

Remarks Note that unlike SCRN, HSCRN does not recognize COLOR.

X must be in the range 0 to 279 and Y must be in the range 0 to 191.

Example

```
10 HGR
15 HCOLOR = 3
20 HPLOT 0,100 TO 279,100
30 PRINT HSCRN (46,100), HSCRN (20,20)
RUN
 -1 0
```

Appendices

Appendix A
Microsoft BASIC and Applesoft: A Comparison

Microsoft BASIC Interpreter Version 5.27 includes many features not found in Applesoft BASIC. In addition, some features that are common to both Microsoft BASIC and Applesoft BASIC work differently, depending on which version of BASIC you have.

By taking note of these differences and using the new features provided by Microsoft BASIC, you can take advantage of increased BASIC programming power.

A.1 Features of Microsoft BASIC Not Found in Applesoft

The following features are found in Microsoft BASIC only. A brief description of these features is given here; for more information on the syntax, purpose, and peculiarities of each feature, see Chapters 2 through 4 of this manual.

CHAIN and COMMON

Used to call in another BASIC program from disk and pass variables to it. This feature allows the disk to be used as program memory.

CALL

Used to call 6502 or Z80 assembly language subroutine or FORTRAN subroutine.

PRINT USING

Greatly enhances programming convenience by making it easy to format output. Includes asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion.

Built-in Disk I/O Statements

Since standard Applesoft BASIC and Integer BASIC were not designed for a disk environment, disk I/O commands have to be included in PRINT statements. With Microsoft BASIC Interpreter 5.27's built-in disk I/O statements, this process is eliminated (you don't need to enter PRINT "CTRL-D").

WHILE/WEND

Gives BASIC a more structured flavor. By putting a WHILE statement in front of a loop and a WEND statement at the end of a loop, you can make BASIC continuously execute the loop as long as a given condition is true.

EDIT Commands

Let you edit individual program lines easily and efficiently without reentering the whole line.

AUTO and RENUM

RENUM makes it easier to edit and debug programs by letting you automatically renumber lines in user-specified increments. AUTO is a convenience feature that generates line numbers automatically after every carriage return.

IF...THEN[...ELSE]

Extends the IF statement to provide for handling the negative case of IF.

ANSI Compatibility

Microsoft BASIC Interpreter Version 5.27 meets the ANSI qualifications for BASIC, as set forth in document BSRX3.60-1978. That means any program you write on your Apple in Microsoft BASIC can be run on any other machine that has an ANSI-standard BASIC.

Compilability

Microsoft has developed a BASIC compiler that compiles Microsoft BASIC Interpreter programs into directly executable Z80 machine code. The compiler is available separately.

Powerful Data Types

Microsoft BASIC Interpreter Version 5.27 has three variable types —fast two-byte true integer variables, single precision variables, and double precision variables—to give it 16-digit precision, as opposed to the 9-digit precision available on the Apple. Also, hexadecimal and octal constants can be used.

Added String Functions

The functions: INSTR, HEX$, OCT$, STRING$, and direct assignment of substrings with MID$ are implemented in Microsoft BASIC.

Added Operators

New Boolean operators AND, OR, XOR, IMP, and EQV are provided with Microsoft BASIC. True integer arithmetic is supported with an integer division and MOD operators.

User-Defined Functions

Microsoft BASIC user-defined functions allow multiple arguments.

Protected Files

Programs can be saved in protected binary format. See "SAVE," Section 3.67.

We have also added four new features to Microsoft BASIC, to take advantage of the Apple's unique characteristics. They are:

BUTTON

A function used to determine whether a paddle button has been pressed.

BEEP

A statement that generates a tone of specified pitch and duration.

HSCRN

A function used to determine if a point has been plotted on the high-resolution screen at a specified point.

VPOS

A function that returns the cursor's vertical position.

A.2 Applesoft Features Supported by Microsoft BASIC

This version of Microsoft BASIC supports low-resolution graphics, sound, cursor control, as well as other Applesoft BASIC features. This version also supports all of the Applesoft high-resolution graphics features except DRAW, XDRAW, SCALE, and ROT.

Applesoft-compatible statements and functions found in GBASIC are listed below.

COLOR	NORMAL
GR	PDL(0)
HCOLOR	PLOT
HGR	POP
HLIN	SCRN
HPLOT	TEXT
HTAB	VLIN
INVERSE	VTAB

A.3 Applesoft Features Used Differently in Microsoft BASIC

Certain Microsoft BASIC statements and commands are used differently than their Applesoft counterparts. You should be aware of these differences when writing Microsoft BASIC programs. Those statements that differ are listed below; for more information see Chapters 2 and 3 of this manual.

CALL
FOR...NEXT
GR
HGR
IF...THEN[...ELSE]
INPUT
ON ERROR GOTO
RESUME
TEXT

A.4 Applesoft Features Not Supported

The following features found in Applesoft BASIC are not supported in Microsoft BASIC.

cassette LOAD	RECALL
cassette SAVE	ROT
DRAW	SCALE
FLASH	screen editing (ESC A, B, C, D)
HIMEM...LOMEM	SHLOAD
IN#	STORE
PR#	XDRAW

Appendix B

Differences Between Microsoft BASIC Interpreter Release 5.27 and Earlier Releases

B.1 Microsoft BASIC 5.27 Features

For the SoftCard version of Microsoft BASIC, we have made a few very minor changes to normal CP/M and Microsoft BASIC features. If you are accustomed to programming in Microsoft BASIC under CP/M, you will want to note the following changes:

TRON/TROFF

Statement name has been changed to TRACE/NOTRACE. Operation of this statement remains the same.

DELETE

DELETE and DEL can be used interchangeably. Operation of this statement remains the same.

WIDTH

The default width is 40 columns for Apple video output and 80 columns for external terminals or 80-column display interface boards.

WAIT

WAIT now monitors the status of an address rather than that of a machine input port. The effect, however, remains the same.

The following statements are not implemented in this version:

CLOAD

CSAVE

NULL

INP

OUT

Note

Microsoft BASIC Interpreter (Version 5.27) programs transferred to the Apple must be in ASCII format (i.e., saved with the A option). They cannot be in binary format.

The execution of BASIC programs written under Microsoft BASIC Interpreter Release 4.51 and earlier can be affected by some of the new features in Releases 5.0 and higher. Before attempting to run such programs, check for the following:

New reserved words: CALL, CHAIN, COMMON, WHILE, WEND, WRITE, OPTION BASE, and RANDOMIZE.

Conversion from floating-point to integer values results in rounding, as opposed to truncation. This affects not only assignment statements (e.g., I% = 2.5 results in I% = 3), but also function and statement evaluations (e.g., TAB(4.5) goes to the 5th position; A(1.5) yields A(2); and X = 11.5 MOD 4 yields 0 for X).

The body of a FOR...NEXT loop is skipped, if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Division by zero and overflow no longer produce fatal errors. See Chapter 2.

The RND function has been changed so that RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers with each RUN, unless RANDOMIZE is used. See Section 3.59, "RANDOMIZE" and Section 4.33, "RND."

The rules for printing single precision and double precision numbers have been changed. See Section 3.55, "PRINT."

String space is allocated dynamically. The first argument in a two-argument CLEAR statement sets the end of memory. The second argument sets the amount of stack space. See Section 3.5, "CLEAR."

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.), or with a carriage return causes the message ?REDO FROM START to be printed. No assignment of input values is made until an acceptable response is given.

Two additional field formatting characters are available for use with the PRINT USING statement. An ampersand (&) is used to designate variable length string fields, and an underscore (_) signifies a literal character in a format string.

If the expression supplied with the WIDTH statement is 255, BASIC uses an "infinite" line width; that is, it does not insert carriage returns. WIDTH LPRINT can be used to set the line width at the line printer. See Section 3.77, "WIDTH."

The at-sign (@) and underscore (_) are no longer used as editing characters.

Variable names are significant up to 40 characters and can contain embedded reserved words. However, reserved words must now be delimited by spaces. To maintain compatibility with earlier versions of Microsoft BASIC, spaces are automatically inserted between adjoining reserved words and variable names.

Warning

This insertion of spaces may cause the end of a line to be truncated if the line length is close to 255 characters.

Microsoft BASIC programs can be saved in a protected binary format. See Section 3.67, "SAVE."

Reserved words must be preceded by and followed by a space.

Appendix C

Converting Programs to Microsoft BASIC

If you have programs written in a BASIC other than Microsoft BASIC, some minor adjustments may be necessary before they can be run with Microsoft BASIC Interpreter. Here are some specific things to look for when converting BASIC programs to Microsoft BASIC.

C.1 String Dimensions

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Microsoft BASIC statement DIM A$(J).

Some BASICs require use of a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for Microsoft BASIC string concatenation.

In Microsoft BASIC, the MID$, RIGHT$, and LEFT$ functions are used to form substrings out of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	Microsoft BASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J−I+1)

If the substring reference occurs on the left side of an assignment operator and X$ is used to replace characters in A$, convert as follows:

Other BASIC	Microsoft BASIC
A$(I)=X$	MID$(A$,1,1)=X$
A$(I,J9)=X$	MID$(A$,I,J−I+1)=X$

C.2 Multiple Assignments

Some BASICs allow statements of the form

```
10 LET B = C = 0
```

to set B and C equal to zero. Microsoft BASIC would interpret the second equal sign as a logical operator and set B equal to −1 if C equalled 0. To ensure compatibility with Microsoft BASIC, convert this statement to two assignment statements:

```
10 C = 0:B = 0
```

C.3 Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With Microsoft BASIC, be sure all statements on a line are separated by a colon (:) instead.

C.4 MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly in Microsoft BASIC.

Appendix D
Microsoft BASIC Disk I/O

Disk I/O procedures for the beginning BASIC user are examined in this appendix. If you are new to BASIC, or if you are encountering disk-related errors, read through these procedures and program examples to make sure you are using all the disk statements correctly.

Note

Refer to Section 2.4, "CP/M File Naming Conventions," to determine how to specify disk files correctly.

D.1 Program File Commands

The following is a review of the commands and statements used in program file manipulation.

Note

The CP/M operating system appends a default extension of .BAS to the filename given in a SAVE, RUN, MERGE, or LOAD command.

SAVE <filespec>[,A]	Writes to disk the program that currently resides in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses compressed binary format.)

LOAD <filespec>[,R]	Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before loading. If R is included, however, open data files are kept open. Thus, programs can be chained or loaded in sections and access the same data files. (LOAD <filespec>,R and RUN <filespec>,R are equivalent.)
RUN <filespec>[,R]	RUN <filespec> loads the program from disk into memory and runs it. RUN deletes the current contents of memory, and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN <filespec>,R and LOAD <filespec>,R are equivalent.)
MERGE <filespec>	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command is executed, the "merged" program resides in memory, and BASIC returns to command level.
KILL <filespec>	Deletes the file from the disk. <filespec> can be a program file or a sequential or random access data file.
NAME <old filespec> AS <filename>	Changes the name of a disk file. NAME can be used with program files, random access files, or sequential files.

D.2 Protecting Files

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

D.3 Disk Data Files: Sequential and Random Access I/O

There are two types of disk data files that can be created and accessed by a BASIC program: sequential files and random access files.

D.3.1 Sequential Files

Sequential files are easier to create than random access files, but are limited in flexibility and speed when it comes to accessing data. The data written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order sent. The data is read back in the same way.

The following statements and functions are used with sequential files in sequential order.

OPEN
PRINT__#
PRINT USING#
WRITE__#
INPUT#
LINE INPUT#
EOF
LOC
LOF
CLOSE

Accessing a Sequential File

The following program steps are required to create a sequential file and access the data in it:

1.	OPEN the file in "O"mode.	`OPEN "O",#1,"DATA"`
2.	Write data to the file using the PRINT# statement. (WRITE# can be used instead.)	`PRINT#1,A$;B$;C$`
3.	To access the data in the file, you must CLOSE the file and reopen it in "I" mode.	`CLOSE #1` `OPEN "I",#1,"DATA"`
4.	Use the INPUT# statement to read data from the sequential file into the program.	`INPUT#1,X$,Y$,Z$`

Program 1 is a short program that creates a sequential file, "DATA," from information you input at the keyboard.

Program 1—Create a Sequential Data File

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$ = "DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
```

```
RUN

 NAME? MICKEY MOUSE
 DEPARTMENT? AUDIO/VISUAL AIDS
 DATE HIRED? 01/12/72

 NAME? SHERLOCK HOLMES
 DEPARTMENT? RESEARCH
 DATE HIRED? 12/03/65

 NAME? EBENEEZER SCROOGE
 DEPARTMENT? ACCOUNTING
 DATE HIRED? 04/27/81
```

```
NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/81

NAME? etc.
```

Now look at Program 2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1981.

Program 2—Accessing a Sequential File

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2) = "81" THEN PRINT N$
40 GOTO 20
RUN

 EBENEEZER SCROOGE
 SUPER MANN
 Input past end in 20
```

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an INPUT PAST END error. To avoid this error, insert line 15, which uses the EOF function to test for the end-of-file,

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The commas at the end of the format string separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was opened. A sector is a 128-byte block of data.

Adding Data to a Sequential File

If you have a sequential file residing on disk and want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents.

The following procedure can be used to add data to an existing file called "NAMES."

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY."
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY."
6. Rename "COPY" as "NAMES" and CLOSE.
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program 3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# reads in characters from the disk until it sees a carriage return (it does not stop at quotation marks or commas) or until it has read 255 characters.

Program 3—Adding Data to a Sequential File

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$ = "" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":
RESUME 120
2010 ON ERROR GOTO 0
```

The error-trapping routine in line 2000 traps a FILE NOT FOUND error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

D.3.2 Random Access Files

Creating and accessing random access files requires more program steps than creating and accessing sequential files. However, there are advantages to using random access files. One advantage is that random access files require less room on the disk, since BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage of using random access files is that data can be accessed randomly, i.e., anywhere on the disk. However, it is not necessary to read through all the information from the beginning of the file, as with sequential files. This is possible because the information is stored and accessed in distinct units called records, each of which is numbered.

The statements and functions that are used with random access files are:

Statements	*Functions*
CLOSE	CVD
FIELD	CVI
GET	CVS
LOC	LOF
LSET	MKD$
OPEN	MKI$
PUT	MKS$
RSET	

Creating a Random Access File

The following program steps are required to create a random access file.

1. OPEN the file for random access ("R"mode). The following example specifies a record length of 32 bytes. If the record length is not specified, the default is 128 bytes.

   Example:

   ```
   OPEN "R", 1,"FILE",32
   ```

2. Use the FIELD statement to allocate space in the random access buffer for the variables that will be written to the random access file.

   Example:

   ```
   FIELD #1, 20 AS N$,
   4 AS A$, 8 AS P$
   ```

3. Use LSET to move the data into the random access buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ to make a single precision value into a string, and MKD$ to make a double precision value into a string.

   Example:

```
LSET N$ = X$
LSET A$ = MKS$(AMT)
LSET P$ = TEL$
```

4. Write the data from the buffer to the disk using the PUT statement.

   Example:

```
PUT #1,CODE%
```

Program 4 takes information that is input at the terminal and writes it to a random access file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Note

Do not use a fielded string variable in an INPUT or LET statement. Doing so causes the pointer for that variable to point into string space instead of the random access file buffer.

Program 4—Create a Random Access File

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$ = X$
80 LSET A$ = MKS$(AMT)
90 LSET P$ = TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Accessing a Random Access File

The following program steps are required to access a random access file:

1. OPEN the file in "R" mode.

   Example:

   ```
   OPEN "R", 1,"FILE",32
   ```

2. Use the FIELD statement to allocate space in the random access buffer for the variables that will be read from the file.

   Example:

   ```
   FIELD #1 20 AS N$,
   4 AS A$, 8 AS P$
   ```

Note

In a program that performs both input and output on the same random access file, you can often use just one OPEN statement and one FIELD statement.

---

3. Use the GET statement to move the desired record into the random access buffer.

   Example:

   ```
   GET #1,CODE%
   ```

4. The data in the buffer can now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

   Example:

   ```
   PRINT N$
   PRINT CVS(A$)
   ```

Program 5 accesses the random access file "FILE" that was created in Program 4. By entering a three-digit code at the keyboard terminal, the information associated with that code is read from the file and displayed.

Program 5—Access a Random Access File

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

The LOC function, when used with random access files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is greater than 50.

Program 6 is an inventory program that illustrates random file access.

Program 6—Inventory

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
"BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
```

```
500 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q% = CVI(Q$) + A%
530 LSET Q$ = MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q% = CVI(Q$)
620 IF (Q% - S%)<0 THEN PRINT"ONLY";Q%;" IN STOCK":
GOTO 600
630 Q% = Q% - S%
640 IF Q% = <CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
" REORDER LEVEL";CVI(R$)
650 LSET Q$ = MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I = 1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART
NUMBER":GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$ = CHR$(255)
930 FOR I = 1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

In this program, the record number is used as the part number. It is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the various inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

Appendix E
Microsoft BASIC Assembly Language Subroutines

The SoftCard version of Microsoft BASIC, like all versions of Microsoft BASIC, contains provisions for interfacing with assembly language subroutines through the USR function and the CALL statement.

E.1 Memory Allocation

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest possible memory location minus the amount of memory needed for the assembly language subroutine(s) by using the /M:switch. BASIC uses all memory available from its starting location upwards, so only the topmost locations in memory can be set aside for assembly language subroutines.

If, when an assembly language subroutine is called, more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine can be loaded into memory through the operating system or the BASIC POKE statement. In addition, routines can be assembled with the Microsoft 8080 Macro Assembler and loaded with the Microsoft 8080 Linking Loader.

E.2 USR Function Calls

The USR function allows assembly language subroutines to be called in the same way BASIC intrinsic functions are called.

The format of the USR function is

USR[<digit>](<argument>)

where <digit> is a number from 0 to 9 and the <argument> is any numeric or string expression. <digit> specifies which USR routine is being called, and corresponds to the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A can be one of the following:

Value in A	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating-point number
8	Double precision floating-point number

If the argument is a number, the HL register pair points to the Floating-Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC+0 contains the lower 8 bits of the argument.

FAC+1 contains the upper 8 bits of the argument.

If the argument is a single precision floating-point number:

FAC+0 contains the lowest 8 bits of mantissa.

FAC+1 contains the middle 8 bits of mantissa.

FAC+2 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

FAC+3 is the exponent minus 128; the binary point is to the left of the most significant bit of the mantissa.

If the argument is a double precision floating-point number:

FAC-4 through FAC-1 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the DE register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

Warning

If the argument is a string in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. For example:

```
A$ = "BASIC" + ""
```

This copies the string literal into string space and prevents alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in the HL register pair as the value of the function, forcing the value returned by the function to be an integer. To execute MAKINT, use the following sequence to return from the subroutine:

```
PUSH H          ;save value to be returned
LHLD xxx        ;get address of MAKINT
                ;routine
XTHL            ;save return on stack and
                ;get back HL
RET             ;return
```

Also, the argument of the function, regardless of its type, can be forced to an integer by calling the FRCINT routine to get the integer value of the argument in the HL register pair. Execute the following routine:

```
LXI H,sub1      ;get address of subroutine
                ;continuation
PUSH H          ;place on stack
LHLD xxx        ;get address of FRCINT
PCHL
    SUB1: . . . . .
```

E.3 CALL Statement

User function calls to Z80 assembly language subroutines can be made with the CALL statement (see "CALL," Section 3.3). A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return with a simple "RET" instruction. (CALL and RET are 8080 assembly language instructions—see an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to three, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).
2. If the number of parameters is greater than 3, they are passed as follows:
   a. Parameter 1 in HL
   b. Parameter 2 in DE
   c. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that with this scheme the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or type of parameters.

If the subroutine expects more than three parameters, and there is a need to transfer them to a local data area, use a system subroutine to perform the transfer. The subroutine $AT (listed in the following paragraphs) is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subroutine is responsible for saving the first two parameters before calling AT. For example, if a subroutine expects five parameters, it should look like the following example:

```
SUBR:   SHLD   P1      ;SAVE PARAMETER 1
        XCHG
        SHLD   P2      ;SAVE PARAMETER 2
        MVI    A,3     ;NO. OF PARAMETERS LEFT
        LXI    H,P3    ;POINTER TO LOCAL AREA
        CALL   $AT     ;TRANSFER THE OTHER 3
                       PARAMETERS
        :
        [Body of subroutine]
        :
        RET            ;RETURN TO CALLER
P1:     DS     2       ;SPACE FOR PARAMETER 1
P2:     DS     2       ;SPACE FOR PARAMETER 2
P3:     DS     6       ;SPACE FOR PARAMETERS 3-5
```

A listing of the argument transfer routine $AT follows.

```
00100   ;              ARGUMENT TRANSFER
00200   ;[B,C]         POINTS TO 3RD PARAMETER.
00300   ;[H,L]         POINTS TO LOCAL STORAGE FOR
                       PARAMETER 3
00400   ;[A]           CONTAINS THE # OF PARAMETERS
                       TO XFER (TOTAL-2)
00500
00600
00700           ENTRY  $AT
00800   $AT:    XCHG           ;SAVE HL IN DE
00900           MOV    H,B
01000           MOV    L,C     ;HL = PTR TO PARA-
                               METERS
01100   AT1:    MOV    C,M
01200           INX    H
01300           MOV    B,M
01400           INX    H       ;BC = PARAMETER ADR
01500           XCHG           ;HL POINTS TO LOCAL
                               STORAGE
01600           MOV    M,C
01700           INX    H
01800           MOV    M,B
01900           INX    H       ;STORE PARAMETER IN
                               LOCAL AREA
02000           XCHG           ;SINCE GOING BACK TO
                               AT1
02100           DCR    A       ;TRANSFERRED ALL
                               PARAMETERS?
02200           JNZ    AT1     ;NO, COPY MORE
02300           RET            ;YES, RETURN
```

When accessing parameters in a subroutine, remember that they are pointers to the actual arguments passed.

Note

You must match the *number*, *type*, and *length* of the arguments in the calling program with the parameters expected by the subroutine. This applies to BASIC subroutines as well as those written in assembly language.

Appendix F
Mathematical Functions

The derived functions that are not intrinsic to Microsoft BASIC can be calculated as follows.

Mathematical Function	Microsoft BASIC Equivalent
Secant	SEC(X)=1/COS(X)
Cosecant	CSC(X)=1/SIN(X)
Cotangent	COT(X)=1/TAN(X)
Inverse sine	ARCSIN(X)=ATN(X/SQR(−X*+1))
Inverse cosine	ARCCOS(X)=−ATN (X/SQR(−X*X+1))+1.5708
Inverse secant	ARCSEC(X)=ATN(X/SQR(X*X−1)) +SGN(SGN(X)−1)*1.5708
Inverse cosecant	ARCCSC(X)=ATN(X/SQR(X*X−1)) +(SGN(X)−1)*1.5708
Inverse cotangent	ARCCOT(X)=ATN(X)+1.5708
Hyperbolic sine	SINH(X)=(EXP(X)−EXP(−X))/2
Hyperbolic cosine	COSH(X)=(EXP(X)+EXP(−X))/2
Hyperbolic tangent	TANH(X)=(EXP(−X)/EXP(X)+EXP (−X))*2+1
Hyperbolic secant	SECH(X)=2/(EXP(X)+EXP(−X))
Hyperbolic cosecant	CSCH(X)=2/(EXP(X)−EXP(−X))
Hyperbolic cotangent	COTH(X)=EXP(−X)/(EXP(X)−EXP (−X))*2+1
Inverse hyperbolic sine	ARCSINH(X)=LOG(X+SQR(X*X+1))

Mathematical Function	Microsoft BASIC Equivalent
Inverse hyperbolic cosine	ARCCOSH(X)=LOG(X+SQR(X*X−1))
Inverse hyperbolic tangent	ARCTANH(X)=LOG((1+X)/(1−X))/2
Inverse hyperbolic secant	ARCSECH(X)=LOG((SQR(−X*X+1)+1)/X)
Inverse hyperbolic cosecant	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X−+1)+1)/X
Inverse hyperbolic cotangent	ARCCOTH(X)=LOG((X+1)/(X−1))/2

Appendix G
Microsoft BASIC Floating-Point Numeric Format

This discussion provides the information needed to encode and decode Microsoft floating-point representation. This information is intended for advanced assembly language programmers, and should not be viewed as an introduction to binary math.

Note that the encoding information presented below pertains only to integral numbers. Encoding fractional numbers is a very complex process. We recommend that you contact the Microsoft Technical Support office if you need help encoding fractional numbers.

G.1 Encoding an Integral Floating-Point Number

Microsoft floating-point representation is a normalized binary approximation of the argument number. It consists of two parts, the mantissa and the exponent.

The mantissa is a 24-bit (single precision) or 56-bit (double precision) normalized approximation of the number. The most significant bit of the mantissa is always assumed to be a 1 after normalization. Therefore, this bit is free to represent the sign of the mantissa.

The exponent is an "excess-80" (80H) representation of the binary (powers of two) exponent of the number. 80H is added to the binary exponent, so that positive exponents are assumed to have an exponent of 80H or greater, while negative exponents are assumed to have an exponent of 7FH or less. An exponent of zero indicates the number itself is zero, regardless of the mantissa.

The procedure for encoding an integral number into floating-point representation consists of four steps:

1. Convert to binary format
2. Normalize
3. Compute the exponent
4. Store

This process is best explained through an example. In the steps explained below, the number 5.00 is converted to a single precision number.

1. The conversion to binary format can be done in many ways. The simplest of these is the subtraction method.

   This method uses repeated subtractions of the powers of two until the number is converted. For the purposes of our example, a partial table of the positive powers of two is shown:

   $2^0=1$
   $2^1=2$
   $2^2=4$
   $2^3=8$
   $2^4=16$
   . . .

   Subtract the largest power of two that produces a positive result or zero. If the result is positive or zero, mark a 1 in the binary equivalent column as shown below. If the result is a negative number, mark a zero in the binary equivalent column. If there is a remainder, repeat the subtraction process with the next power of two.

For example:

Conversion	*Binary equivalent*
5−4=1	1

4 (2^2) is the largest number that can be subtracted from 5. The result is a remainder of 1.

Now, see if the next power of two (2^1) can be subtracted from the remainder.

Conversion	*Binary equivalent*
1−2=−1	01

Since 1−2 produces a negative number, do not subtract. Instead, mark a zero.

Repeat the subtraction process with the next largest power of two (2^0=1).

Conversion	*Binary equivalent*
1−1=0	101

One will subtract evenly, so the final binary result is 101.

Note

If you get to the point of subtracting 1 and the result is not zero, you have made an error.

---

2. Now the binary number must be normalized. This is accomplished by moving the *binary point* (the binary equivalent of the decimal point) to the left until it is immediately left of the leftmost 1 of the number (the most significant bit); as the point is moved, count the number of "shifts" that were made. Thus, 101.00... becomes .10100....

The next step in normalization is converting the most significant bit into the sign bit. Because Microsoft floating-point representation assumes that the most significant bit is 1 (this is why the number is normalized), this bit represents the sign of the number. Since the original number was positive, the sign bit becomes zero (1 indicates negative). Therefore, the normalized number is .0010 0000....

3. To convert the number to its final form, calculate the exponent by adding 80H to the number of shifts performed during normalization. Since the binary point was shifted 3 places, add 3. This results in an exponent of 83H. The floating-point number is, therefore, .0010 0000 0000 0000 0000 0000 with an exponent of 83H, or 00 00 20 83 in Hex.

4. The floating-point number is stored as LSB (Least Significant Byte), NSB (Next Significant Byte), MSB (Most Significant Byte), and EXP (Exponent), with LSB stored in low memory and EXP stored in high memory. This is the form presented by a USR function call or a CALL statement.

G.2 Decoding an Integral Floating-Point Number

To decode an integral floating-point number, simply perform the above steps in reverse: Find the most significant bit of the mantissa, check it for sign, set it, and denormalize. For example, the following steps are required to decode 00 00 20 83 Hex:

1. Check the most significant bit of the Most Significant Byte (MSB) for the sign of the number. In this case, the MSB is 0010 0000, so the sign of the number is positive.

2. Set the Most Significant Bit to 1. This results in a binary number of 1010 0000.

3. Denormalize the number by shifting the binary point the necessary number of places. 83H implies shifting the binary point 3 places right, giving us 101.00000, or 5 decimal.

G.3 Decoding a Fractional Floating-Point Number

If the number to be converted is a fraction, it will have a negative exponent.

A negative exponent (7F or less) simply implies that the binary point is shifted to the left instead of the right when decoding. Therefore, 1010 0000 with an exponent of 7DH would become .0001 0100 after denormalization. Because the sign bit was set, we know the original number was negative. Computing from the negative powers of two, we have $2^{-4}+2^{-6}=.0625+.015625=$ .078125. Since the sign of the number is negative, the final result is $-.078125$.

Appendix H
ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	_
010	0AH	LF	053	35H	5	096	60H	`
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH	|
039	27H	'	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal Hex=hexadecimal (H) CHR=character
LF=Line Feed FF=Form Feed CR=Carriage Return
DEL=Rubout

Appendix I

Microsoft BASIC Reserved Words

The following is a list of reserved words used in Microsoft BASIC.

ABS	DIM	INPUT	NOTRACE	SIN
AND	EDIT	INPUT#	OCT$	SPACE$
ASC	ELSE	INPUT$	ON	SPC
ATN	END	INSTR	OPEN	SQR
AUTO	EOF	INT	OPTION	STOP
BEEP	EQV	INVERSE	OR	STR$
BUTTON	ERASE	KILL	PDL	STRING$
CALL	ERL	LEFT$	PEEK	SWAP
CDBL	ERR	LEN	PLOT	SYSTEM
CHAIN	ERROR	LET	POKE	TAB
CHR$	EXP	LINE	POP	TAN
CINT	FIELD	LIST	POS	TEXT
CLEAR	FILES	LLIST	PPRINT	THEN
CLOSE	FIX	LOAD	PRINT	TO
COLOR	FOR	LOC	PRINT#	TRACE
COMMON	FRE	LOF	PUT	USING
CONT	GET	LOG	RANDOMIZE	USR
COS	GOSUB	LPOS	READ	VAL
CSNG	GOTO	LPRINT	REM	VARPTR
CVD	GR	LSET	RENUM	VLIN
CVI	HCOLOR	MERGE	RESET	VPOS
CVS	HEX$	MID$	RESTORE	VTAB
DATA	HGR	MKD$	RESUME	WAIT
DEFDBL	HOME	MKI$	RIGHT$	WEND
DEFINT	HPLOT	MKS$	RND	WHILE
DEFSNG	HTAB	MOD	RSET	WRITE
DEFSTR	IF	NAME	RUN	WRITE#
DEF FN	IMP	NEW	SAVE	XOR
DEF USR	INKEY	NORMAL	SCRN	
DELETE	INP	NOT	SGN	

Appendix J
Error Codes and Error Messages

This Appendix lists Microsoft BASIC error messages.

Table J.1. Operational Errors

Error Code	Message	Explanation
1	NEXT WITHOUT FOR	A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	SYNTAX ERROR	A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).
3	RETURN WITHOUT GOSUB	A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	OUT OF DATA	A READ statement is executed when there are no DATA statements with unread data remaining in the program.

Error Code	Message	Explanation
5	ILLEGAL FUNCTION CALL	A parameter that is out of range is passed to a math or string function. This error may also occur as the result of: 1. A negative or unreasonably large subscript. 2. A negative or zero argument with LOG. 3. A negative argument to SQR. 4. A negative mantissa with a non-integer exponent. 5. A negative record # on a GET or PUT statement. 6. A call to a USR function for which the starting address has not yet been given. 7. An improper argument to MID$, LEFT$, RIGHT$, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.
6	OVERFLOW	The result of a calculation is too large to be represented in Microsoft BASIC's number format. If underflow occurs, the result is zero and execution continues without an error.
7	OUT OF MEMORY	A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.
8	UNDEFINED LINE xxxxx IN yyyyy	A line referenced in a GOTO, GOSUB, IF...THEN [...ELSE], or DELETE statement does not exist.
9	SUBSCRIPT OUT OF RANGE	Caused by one of three conditions: 1. An array element is referenced with a subscript that is outside the dimensions of the array. 2. An array element is referenced with the wrong number of subscripts. 3. A subscript was used on a variable that is not an array.

Error Code	Message	Explanation
10	REDIMENSIONED ARRAY	Caused by one of three conditions: 1. Two DIM statements are given for the same array. 2. A DIM statement is given for an array after the default dimension of 10 has been established for that array. 3. An OPTION BASE statement has been encountered after an array has been dimensioned by either default or a DIM statement.
11	DIVISION BY ZERO	Caused by one of two conditions: 1. A division by zero operation is encountered in an expression. Machine infinity with the sign of the numerator is supplied as the result of the division. 2. The operation of raising zero to a negative power occurs. Positive machine infinity is supplied as the result of the exponentiation, and execution continues.
12	ILLEGAL DIRECT	A statement that is illegal in direct mode is entered as a direct mode command. For example, DEF FN.
13	TYPE MISMATCH	A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa. This error can also be caused by trying to SWAP single precision and double precision values.
14	OUT OF STRING SPACE	String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.
15	STRING TOO LONG	An attempt was made to create a string more than 255 characters long.
16	STRING FORMULA TOO COMPLEX	A string expression is too long or too complex. The expression should be broken into smaller expressions.

Error Code	Message	Explanation
17	CAN'T CONTINUE	An attempt is made to continue a program that: 1. Has halted due to an error. 2. Has been modified during a break in execution. 3. Does not exist.
18	UNDEFINED USER FUNCTION	A USR function is called before the function definition (DEF statement) is given.
19	NO RESUME	An error-handling routine is entered, but it contains no RESUME statement.
20	RESUME WITHOUT ERROR	A RESUME statement is encountered before an error-trapping routine is entered.
21	UNPRINTABLE ERROR	An error message is not available for the error condition which exists. This is usually caused by an ERROR statement with an undefined error code.
22	MISSING OPERAND	An expression contains an operator without a following operand.
23	LINE BUFFER OVERFLOW	An attempt has been made to input a line that has too many characters.
26	FOR WITHOUT NEXT	A FOR statement was encountered without a matching NEXT statement.
29	WHILE WITHOUT WEND	A WHILE statement was encountered without a matching WEND statement.
30	WEND WITHOUT WHILE	A WEND statement was encountered without a matching WHILE statement.

Table J.2. Disk Errors

Error Code	Message	Explanation
50	FIELD OVERFLOW	A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random access file.
51	INTERNAL ERROR	An internal malfunction has occurred in Microsoft BASIC. Report to Microsoft the conditions under which the message appeared.
52	BAD FILE NUMBER	A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
53	FILE NOT FOUND	A FILES, LOAD, NAME, or KILL command or OPEN statement references a file that does not exist on the current disk.
54	BAD FILE MODE	An attempt was made to: 1. Use PUT, GET, or LOF with a sequential file. 2. LOAD a random access file. 3. Execute an OPEN statement with a file mode other than I, O, or R.

Error Code	Message	Explanation
55	FILE ALREADY OPEN	A sequential output mode OPEN is issued for a file that is already open or a KILL is given for a file that is open.
57	DISK I/O ERROR	An I/O error occurred during a disk I/O operation. It is a fatal error; i.e., the operating system cannot recover from the error.
58	FILE ALREADY EXISTS	The filename specified in a NAME statement is identical to a filename already in use on the disk.
61	DISK FULL	All disk storage space is in use.
62	INPUT PAST END	An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.
63	BAD RECORD NUMBER	In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.
64	BAD FILE NAME	An illegal form is used for the filespec with a LOAD, SAVE, or KILL command or an OPEN statement (e.g., a filename with too many characters).
66	DIRECT STATEMENT IN FILE	A direct statement is encountered while loading an ASCII-format file. The LOAD operation is terminated.
67	TOO MANY FILES	An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

Index

MICROSOFT®
10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem (Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

DIGITAL RESEARCH LICENSE INFORMATION

CAREFULLY READ ALL THE TERMS AND CONDITIONS OF THIS AGREEMENT PRIOR TO BREAKING THE DISKETTE SEAL. BREAKING THE SEAL INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS.

IMPORTANT: Our license with Digital Research for the CP/M® Operating System requires that each purchaser of the SoftCard™ with CP/M register with Microsoft® Corporation so that records can be maintained of all CP/M owners. This requirement is made by Digital Research, not by Microsoft, and a post-card is enclosed for reply. THE SERIAL NUMBER ON THE CARD IS THE NUMBER STAMPED ON THE DISK LABELS.

SOFTWARE LICENSE AGREEMENT

IMPORTANT: All Digital Research programs are sold only on the condition that the purchaser agrees to the following license. READ THIS LICENSE CAREFULLY. If you do not agree to the terms contained in this license, return the packaged diskette UNOPENED to your distributor and your purchase price will be refunded. If you agree to the terms contained in this license, fill out the REGISTRATION information and RETURN by mail to Microsoft Corporation.

DIGITAL RESEARCH agrees to grant and the Customer agrees to accept on the following terms and conditions nontransferable and nonexclusive license to use the software program(s) (Licensed Programs) herein delivered with this agreement.

1. TERM: This agreement is effective from the date of receipt of the above-referenced program(s) and shall remain in force until terminated by the Customer upon one month's prior written notice, or by Digital Research as provided below.

Any license under this Agreement may be discontinued by the Customer at any time upon one month's prior written notice. Digital Research may discontinue any license or terminate this Agreement if the Customer fails to comply with any of the terms and conditions of this Agreement.

2. LICENSE: Each program license granted under this Agreement authorizes the Customer to use the Licensed Program in any machine readable form on any single computer system (referred to as System). A separate license is required for each System on which the Licensed Program will be used.

This Agreement and any of the licenses, programs or materials to which it applies may not be assigned, sublicensed or otherwise transferred by the Customer without prior written consent from Digital Research. No right to print or copy, in whole or in part, the Licensed Programs is granted except as hereinafter expressly provided.

3. PERMISSION TO COPY OR MODIFY LICENSED PROGRAMS: The Customer shall not copy, in whole or in part, any Licensed Programs which are provided by Digital Research in printed form under this Agreement. Additional copies of printed materials may be acquired from Digital Research.

Any Licensed Programs which are provided by Digital Research in machine readable form may be copied, in whole or in part, in printed or machine readable form in sufficient number for use by the Customer with the designated System, to understand the contents of such machine readable material, to modify the Licensed Program as provided below, for back-up purposes, OR FOR ARCHIVE PURPOSES, provided, however, that no more than five (5) printed copies will be in existence under any license at any one time without prior written consent from Digital Research. The Customer agrees to maintain appropriate records of the number and location of all such copies of Licensed Programs. The original, and any copies of the Licensed Programs, in whole or in part, which are made by the Customer shall be the property of Digital Research. This does not imply, of course, that Digital Research owns the media on which the Licensed Programs are recorded. The Customer may modify any machine readable form of the Licensed Programs for his own use and merge it into other program material to form an updated work, provided that, upon discontinuance of the license for such Licensed Program, THE LICENSED PROGRAM SUPPLIED BY DIGITAL RESEARCH WILL BE COMPLETELY REMOVED FROM THE UPDATED WORK. ANY PORTION OF THE LICENSED PROGRAM INCLUDED IN AN UPDATED WORK SHALL BE USED ONLY IF ON THE DESIGNATED SYSTEM AND SHALL REMAIN SUBJECT TO ALL OTHER TERMS OF THIS AGREEMENT.

The Customer agrees to reproduce and include the copyright notice of Digital Research on all copies, in whole or in part, in any form, including partial copies of modifications, of Licensed Programs made hereunder.

4. PROTECTION AND SECURITY: The Customer agrees not to provide or otherwise make available any Licensed Program including but not limited to program listings, object code and source code, in any form, to any person other than Customer or Digital Research employees, without prior written consent from Digital Research, except with the Customer's permission for purposes specifically related to the Customer's use of the Licensed Program.

5. DISCONTINUANCE: Within one month after the date of discontinuance of any license under this Agreement, the Customer will furnish Digital Research a certificate certifying that through his best effort, and to the best of his knowledge, the original and all copies, in whole or in part, in ANY form, including partial copies in modifications, of the Licensed Program received from Digital Research or made in connection with such license have been destroyed, except that, upon prior written authorization from Digital Research, the Customer may retain a copy for archive purposes.

6. DISCLAIMER OF WARRANTY: Digital Research makes no warranties with respect to the Licensed Programs. The sole obligation of Digital Research shall be to make available all published modifications or updates made by Digital Research to Licensed Programs which are published within one (1) year from date of purchase, provided Customer has returned the Registration Card delivered with the Licensed Program.

7. LIMITATION OF LIABILITY: THE FOREGOING WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL DIGITAL RESEARCH BE LIABLE FOR CONSEQUENTIAL DAMAGES EVEN IF DIGITAL RESEARCH HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

8. GENERAL: If any of the provisions, or portions thereof, of this Agreement are invalid under any applicable statute or rule of law, they are to that extent to be deemed omitted.